Polishing Design Flaw Definitions

Daniel Speicher, B-IT, Universität Bonn, dsp@bit.uni-bonn.de

> In all methods *M* of class *C*, you may use only methods and fields of the following types and their supertypes: *C itself*, types of *fields* of *C*, types of *parameters* of *M*, or classes that are *instantiated* in *M*.

Figure 1: LAW OF DEMETER, [3]

Every single false positive that an automated design flaw detection tool presents to a developer places a cognitive burden on him and should thus be avoided. A preliminary yet already quite comprehensive catalog of false positives was recently presented in [1]. We elaborated in [5, 4] that developers accept design flaws in the context of specific design ideas, e.g. FEATURE ENVY in the context of the VISITOR pattern. We suggested that it should be possible to incorporate such knowledge into operational design flaw definitions. Inspired by [2] we are convinced that our understanding of the phenomena "design flaw", "design idea" and their conflict could still benefit from an exploratory case study guided by the following research questions: *What is the nature of design ideas that are related to design flaws? How are both related? What are the consequences for operational definitions of design ideas and design flaws?*

1 The exploratory case study

In the hope to generate many false positives we apply the rather rigid classical design flaw LAW OF DEMETER to the thoughtfully designed classical framework JHOTDRAW 5.1. The LAW OF DEMETER is sometimes presented as the motto "Only talk to your friends." - meaning that methods should only access members of certain "friend" types. See Figure 1 for our translation of a C++ version to Java[1]. JHOTDRAW is a framework for graphical drawing editor applications developed by Erich Gamma and others[2]. To assess the severeness of a potential design flaw, we review the negative effect of not following the LAW on *encapsulation*, *coupling*, and *understandability*. We review as well the positive effects of the two possible refactorings of *pushing* parts of the method that access the "stranger" *back* to a "friend" or *lifting* accesses to "stranger"'s members *forward* to a "friend".

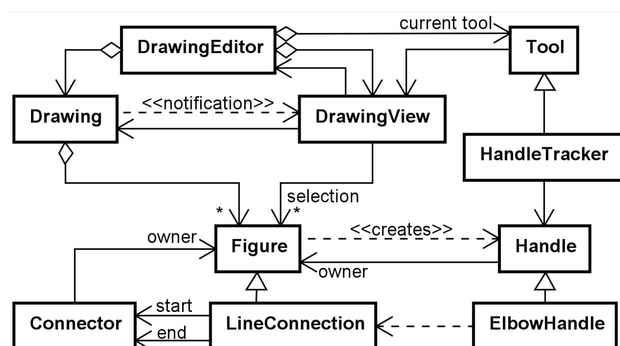

Figure 2: Partial class diagram for JHotDraw 5.1

2 Examples of true and false positives

Let us list five potential design flaws.The first three are in the method `constrainX(int)` in `ElbowHande`, which constrains the movement of a `LineConnection`:

```
private int constrainX(int x) {
    LineConnection line = ownerConnection();
    Figure startFigure = line.start().owner();
    Rectangle start = startFigure.displayBox();
    Insets i1 = startFigure.connectionInsets();
    int r1x = start.x + i1.left;
    int r1width = start.width - i1.left - i1.right-1;
    return Geom.range(r1x, r1x + r1width, x);
}
```

(F1) "`line.start()`" is a false positive since `ElbowHandle`'s only purpose is to manipulate its `LineConnection`. This does not show in the type of a field but in the type of a constructor parameter.[3] **(T1)** "`start().owner()`" is a true positive since there is no need to know the class `Connector` here. There is already a method `getStartFigure()` that lifts the figure access forward. **(F2)** "`start.x`" and all accesses to fields of the class `Rectangle` should be considered to be false positives as discussed in the next section. **(F3)** Accesses to `Figure` after retrieving the selection from `DrawingView` are false positives since the indirection is a consequence of the reasonable global design decision to maintain the `Figure` selection in one place. **(T2)** Comments claim that `DrawingEditor` is a MEDIATOR. It should thus hide the types of COLLEAGUES, but it exposes them through accessors. Potential design flaws in this context should be considered true.

[1] As the corresponding C++ version in [3] our Java version is based on classes and not on objects. It is "strict" - meaning that types of inherited fields are not considered to be "friends".

[2] Dirk Riehle made the source and documentation available in the context of his dissertation at `http://dirkriehle.com/computer-science/research/dissertation/appendix-e.html` .

[3] As soon as we consider `LineConnection` a friend its supertype `Figure` must be considered a friend as well. Thus the two method calls on `startFigure` are no violations anymore.

3 Example review of a potential flaw

As for (F2) we suggest to consider all the 366 accesses to `Rectangle` that are potential design flaws to be false positives. `Rectangle` should be seen as "everybody's friend", i.e. every method should be allowed to access their members. The concept of a rectangle is part of our elementary common knowledge so that the usage of `Rectangle` is easy to understand. Coupling to this class does not create additional maintenance effort, since its structure is shallow and the class is very stable[4]. The lifting refactoring is not an option since this would mean to replace every method returning one `Rectangle` with four methods returning an `int`. In case of (F2) pushing the whole calculation back into the class `Figure` could be considered. This would indeed improve the encapsulation of `Figure`. Nevertheless, calculating the possible coordinates for the ends of a `LineConnection` is a cohesive responsibility that fits well into the `ElbowHandle`. It is not to expect that the same functionality in `Figure` would be used by other classes. Overall, we suggest that the current code is in this respect good enough and that the same is true for other DATA CLASSES as well.

4 Amendments based on design ideas

For (F1) we suggested to consider constructor parameters to play the same role as fields. We can make similar amendments with respect to parameters and instantiations. For (F2) we suggested to consider DATA CLASSES to be "everybody's friend". For (F3) we suggest to consider the methods returning the selection to be an "introducing member", meaning they introduce `Figure` as "friend" to the calling method. Finally we found reasons to adapt the meaning of "type of" and once for recursion. The review of the potential design flaws led us to make use of design ideas (and language concepts) to *complete* the design flaw definition. Some facets of Java needed to be covered, e.g. for anonymous inner classes the friends of the enclosing method should be as well considered its friends. Some type information was not immediately enough available. Creational patterns are considered as alternative ways of instantiation. But, there were as well real *conflicts* between the design flaw definition and design ideas. We expected design ideas, for which we would be willing to *trade* the design flaw as we had discussed in [4, 5]. We found only (F2) and (F3), the easily understandable DATA CLASSES and the reasonable choice to localize the information about the selected figures in the `DrawingView`. We found cases, where we had to *resign*, because the code that needed to change was not under our control or followed very established conventions, that one might not want to change for the specific context. The case (T2) of the undutiful mediator `DrawingEditor` exposing COLLEAGUES needs very well improvement, but is too deeply entrenched in the current design to be solved with isolated refactorings. Therefore one could *postpone* the decision to refactor until a major redesign.

5 From generic to specific design ideas

The design ideas that we discovered range from very generic ("anonymous inner class") to very specific ("`DrawingView` holds `Figure` selection"). Our confidence ranges from very high ("creation method access is like instantiation") to high enough ("data classes are everybody's friend"). After a thorough discussion of the 1215 potential design flaws 67 remain as true positives and 3 remain undecided. Thus the confrontation of a thoughtfully designed software system with a rigid design flaw definition did indeed result as intended in many false positives and the very low precision 67/1215 = 5.5%. Taking more and more specific design ideas into account we can increase this precision: With respect to Java concepts 472 of the 1215 potential flaws can be identified as false [5]. Allowing some interpretative concepts based on Java (thus slightly reduced confidence), we can reduce the number by further 149 cases[6]. Further 499 cases can be excluded taking general object oriented design ideas like creational patterns and DATA CLASS into account[7]. Finally, further 25 cases can be identified as false positives based on specific design ideas in JHotDraw 5.1[8]. We reached a precision of 67/(1215-472-149-499-25) = 67/70 = 95.7%.

References

[1] F. Arcelli Fontana, J. Dietrich, B. Walter, A. Yamashita, and M. Zanoni. Antipattern and code smell false positives: Preliminary conceptualization and classification. In *SANER 2016. Osaka, Japan*, 2016.

[2] S. Easterbrook, J. Singer, M.-A. Storey, and D. Damian. Selecting Empirical Methods for Software Engineering Research. In *Guide to Advanced Empirical Software Engineering*, pages 285–311. Springer London, 2008.

[3] K. J. Lieberherr and I. M. Holland. Formulations and Benefits of the Law of Demeter. *ACM SIGPLAN Notices*, 24(3):67–78, 1989.

[4] D. Speicher. Code Quality Cultivation. In *IC3K 2011, Revised Selected Papers*. Springer Berlin Heidelberg, 2013.

[5] D. Speicher and S. Jancke. Smell Detection in Context. In *WSR 2010, Softwaretechnik-Trends, Band 30, Heft 2*, 2010.

[4] The class is part of the package `java.awt`. Its simplicity makes changes improbable. The mere amount of code depending on it makes changes to it too expensive, i.e. it is as well stable in the sense of R. C. Martin.

[5] Everybody's friends: Collections types (137), `java.lang` types (128), the array "field" `length` (8), `public static` members (270). Friends of the enclosing method of an anonymous inner class are friends of any method in this inner class (23).

[6] Like fields: Constructor parameters (86) - see (F1), Inferred type in collection field (40). Like parameters: Immediate downcasts of parameters. Introducing members: Designated accessors in the JDK (5), `System.out` (11)

[7] Like instantiation: Call to FACTORY METHOD (1) or CREATION METHOD (2), access to SINGLETON (24). Everybody's friend: DATA CLASS (540) - see (F2).

[8] Taking specific types into account: COMPOSITES with specific COMPONENTS (5), `Clipboard` contains `Figures` (2), collection parameters containing `Figures` (3). Introducing members: Accessors to the selected `Figures` in `DrawingView` (20) - see (F3).