Smells in Java Code: Do You Recognize Them?

A look at ten common code smells, or code that violates design principles, in Java. Recognize any in your applications?

 by Ganesh Samarthyam

Smells are structures in code that violate design principles and negatively impact quality [1]. Here are some of the bad smells in Java code.

1. Constant Interface. Constant interfaces have only static final data members declared in them without any methods. Suggested refactoring for this smell depends on kind of constants present in the constant interface: the constants can get added as members in the class or can be rewritten as enums. The interface java.io.ObjectStreamConstant is an example for this smell.

2. Global Variable Class. The class has one public static (non-final) field. Since its freely available for modification by anyone, it becomes an equivalent to a C-like global variable, with only difference that the variable name is prefixed by the class name, thus avoiding name-clashes. Here is an example:

```
class Balls {
    public static long balls = 0;
}
```

3. Global Function Class. It is a public class that has only one **static** public method and no other fields or methods; it can have an optional private constructor to disallow instantiation. It is equivalent to a C-like global function, except that the function needs to be prefixed by the class name to access the function.

4. Publicly Exposed Fields. It is a public class with public non-final, non-static data members and no methods (with an optional constructor). It is difficult to maintain public, C-like classes, as *Effective Java* [2] notes: "Several classes in the Java platform libraries violate the advice that public classes should not expose fields directly. Prominent examples include the Point and Dimension classes in the java.awt package. Rather than examples to be emulated, these classes should be regarded as cautionary tales."

5. Orphan Abstract. An abstract class does not have any concrete derived classes. An abstract class can be used meaningfully only when implemented by concrete derived classes. If the abstract class is not useful, it can be removed. If the class can be instantiated consider, making it concrete. If it represents an useful abstraction, provide one or more concrete classes that implement that abstract class.

6. Forgotten Interface. A class implements all the methods with the same signatures as the methods listed in an interface. It is a likely mistake that the class intended to implement an interface, but forgot to list the interface as its base type. A consequence of this smell is that the objects of the class cannot be treated as subtype of the interface and hence the benefit of subtyping and runtime polymorphism is not exploited.

7. Clone Class. A Clone class is an exact replica of another class (unrelated by inheritance). Essentially, only the name of the class is different, but all its members, their signature, accessibility, etc. are the same. A constraint is that at least one member should be present in the class. An exception is that order of its members might be different.

If two or more classes have common data and behavior, they should inherit from a common class that captures those data and behaviors. So, one fix is to check if it is possible to provide a common base class and make the clone classes as derived classes. Clone classes often occur because of copy-pasted code; in that case, it is better to remove such duplicate classes and instead use a unique class.

8. Lonely Class. The class does not depend on or use any other class; also, no other classes depend on or use the class. This might not really be a mistake or error in design; however, it is rare to see a very independent class that neither uses other classes nor used by any other classes. So, it is possible that it is a design mistake.

9. Abstract Leaf. Abstract classes are most meaningful and useful when we have them near the roots of a hierarchy. In another extreme, it is meaningless to have them as leaves in a hierarchy: they are useless and can be eliminated; abstract leaf classes usually indicate a design mistake.

10. Tagged Class. This smell often occurs in programs written by programmers from structured programming background. They often define classes like structs and instead of providing an inheritance hierarchy of related types and use runtime polymorphism, they often have an enumeration to list types and have switch-case statements (or chained if-else statements) to distinguish between the types to do type specific operations.

The term 'tag class' is from "Effective Java" [2]: "it is a class whose instances come in two or more flavors and contain a tag field indicating the flavor of the instance".

A detect a tag class in code, a check is made for the following features:

- An enumeration (or public static final ints) to indicate the flavour of the instance
- A field to store the value of the enum/int; typically, the constructor sets this field
- A switch statement in one or more methods which execute code based on the tag value

Obvious refactoring of the tagged class is a class hierarchy.

Do you recognize them from your experience? What are the smells that you commonly see in Java code that are missing in this list?

There are numerous static analyzer tools (FindBugs, PMD, etc) available for detecting "bug patterns" in Java. However, it is surprising that there aren't many tools available that are dedicated to detecting such code smells. Hope some tools will emerge in near future.

References:

[1] “Refactoring for Software Design Smells: Managing Technical Debt”, Girish Suryanarayana, Ganesh Samarthyam, Tushar Sharma, ISBN - 978-0128013977, Morgan Kaufmann/Elsevier, 2014.

[2] "Effective Java" (2nd Edition), Joshua Bloch, Addison-Wesley, 2008.