Towards Variability-Aware Smells

Redundant Annotation

DEFINITION:

`Redundant Annotation` variability-aware smell is characterized when one code portion (e.g. class, method, or code block) is redundantly annotated, in a nested fashion, with the same feature tag name (same feature).

EXAMPLE:

Listing 2 shows examples of this smell for the features *AStorage*, *LLStorage* and Locking. Redundantly, the features *AStorage (${AStorage}*) and *LLStorage (${LLStorage}*) have been annotated on the top of the class Stack (line 1) and on the attributes definition (lines 3 and 9) before the first feature tag annotation be closed (`#endif directive`). The same issue happens on lines 1 and 9 where feature *Locking (${Locking}*) is redundantly annotated.

Listing 2: Redundant annotation example.

```
1  //#if ${AStorage} == "T" or ${LLStorage} == "T" or ${Locking} == "T"
2  class Stack <E> {
3   List <E> store = new List<E>();
4   //#if ${AStorage} == "T"
5   List <E> store = new ArrayList<E>();
6  //#endif
7  //#if ${LLStorage} == "T"
8    List <E> store = new LinkedList<E>();
9  //#endif
10 //#if ${Locking} == "T"
11   public void push(E e, Lock lock) {
12    lock.lock();
13    store.add(e);
14    lock.unlock();
15  }
16   E pop(Lock lock) {
17    lock.lock();
18    try { return store.remove(store.size()-1); }
19     finally { lock.unlock(); }
20   }
21 //#endif
22 }
23 //#endif
```

PROBLEM:

`Redundant Annotation` impact directly on maintenance and evolvability, because when the SPL developer needs to change a feature implementation redundantly annotated, the SPL developer needs to know about the `redundant annotation` to avoid do not apply the changes in the whole feature implementation. A harmful situation also can happen if, in the product derivation process, the SPL developer or product manager do not set as true or false all annotations which identify a feature implementation in a class.