Code Smells in Software Product Lines

Wolfram Fenske[1] and Sandro Schulze[2]
`wfenske@ovgu.de` `sanschul@tu-braunschweig.de`

[1]University of Magdeburg, Germany
[2]TU Braunschweig, Germany

Introduction

This document is supplementary to a survey we send to participants of the FOSD meeting 2014 about code smells in software product lines. It contains the description of six code smells that take variability into account as first-class concept, leading to *variability-aware code smells*. Although a comprehensive empirical study is missing for most of these code smells, we argue that these smells may impede evolution and maintenance of SPLs.

For each code smell, we denote the original code smell together with its description (based on Fowler's refactoring book). Then, we provide a description of the variability-aware smell and why we think that it is a smell. Moreover, we mention whether this smell is independent of the variability mechanism or specific to composition-based or annotation-based variability[1]. Finally, we provide an example for the variability-aware code smell, either by an exemplary code fragment or, if this is not possible for some reason, by an exemplary scenario.

1 Inter-Feature Code Clones

Derived from: DUPLICATED CODE

Original description

> Number one in the stink parade is duplicated code. If you see the same code structure in more than one place, you can be sure that your program will be better if you find a way to unify them.

Variability-Aware Description: Duplicated code in software product lines can appear in two variaties: First, code may be duplicated within a feature. We call these *intra-feature* code clones. They are similar in nature to code clones in traditional single-system engineering. Consequently, they are not interesting from an SPL point of view.

More interesting is the second case, when code is duplicated across different features. That is, there are two or more features in the product line which contain

[1] Note, that we currently focus on FOP for composition-based and C preprocessor for annotation-based variability

similar code. This is the smell we call Inter-Feature Code Clones. As a result of this duplication, maintenance (e.g., consistent change of duplicated code fragments) may be hindered.

Applies to: Annotation-based and composition-based implementation techniques

Example: Listing 1.1 shows an example from the Graph Product Line (GPL) (Lopez-Herrejon and Batory, GCSE 2001). The GPL uses feature-oriented programming (FOP). In the example, both, feature BFS and DFS, contain an exact clone of the `search` method.

Listing 1.1. Code Clones in the Graph Product Line

```
// Feature 'BFS'
public class Graph {
  public void search(Workspace w) {
    VertexIter vxiter = getVertices();
    if (vxiter.hasNext() == false) return;
    while (vxiter.hasNext()) {
      Vertex v = vxiter.next();
      v.init_vertex(w);
    }
    /* more source code... */
  }
  /* more source code... */
}

// Feature 'DFS'
public class Graph {
  public void search(Workspace w) {
    VertexIter vxiter = getVertices();
    if (vxiter.hasNext() == false) return;
    while (vxiter.hasNext()) {
      Vertex v = vxiter.next();
      v.init_vertex(w);
    }
    /* more duplicated source code... */
  }
  /* more source code... */
}
```

2 Long Refinement Chain

Derived from: LONG METHOD

Original description

> Since the early days of programming people have realized that the longer a procedure is, the more difficult it is to understand.

Variability-Aware Description: If a method is refined by many features, it is hard to understand what the effective method implementations may look like in different configurations. Moreover, creating a new refinements is difficult, because it is not obvious which information the new refinement is built on (since this is configuration-specific). Hence, understanding an frequently refined method is difficult and thus, evolution (e.g., extending this method) may be influenced in a negative way.

Applies to: Composition-based implementation techniques

Examples: In the GUIDSL, method `process(Model)` of class `Main` is introduced as an empty stub by feature dmain. It is subsequently refined by five other features (fillgs, propgs, formgs, clauselist, modelopts), whereas the average refinement depth is lower than one (i.e., most of the methods are never refined). Each refinement contains a call to `original()` and between three to nine additional lines of code. Most of these refinements can occur in different combinations, depending of the feature selection. Hence, when extending one of these methods or adding a new refinement, you must be aware of all existing refinements and possible side effects of changing or adding code.

3 Annotation Bundle

Derived from: Long Method

Original description

> Since the early days of programming people have realized that the longer a procedure is, the more difficult it is to understand.

Variability-Aware Description: Annotation Bundle is a smell very similar to Long Refinement Chain, but applies to annotation-based implementation techniques. It describes a method that contains an overly large amount of annotated code, compared to the total lines of code (of this method). Moreover, many different preprocessor directives/expressions are used (maybe even nested), meaning that many different features control the presence or absence of the annotated statements. Hence, it is difficult to understand the method for a certain configuration or in its entirety. Similar to Long Refinement Chain, this may impede the comprehension of the code and thus, hinder evolution.

Example: In the following listing, Listing 1.2, we show an example from Firefox version 28, where C preprocessor is heavily used to annotate variable parts of the method implementation.

Listing 1.2. Firefox: memory/mozjemalloc/jemalloc.c
Excerpt from `bool malloc_init_hard(void)`

```
/* More code ... */
for (k = 0; k < nreps; k++) {
        switch (opts[j]) {
        case 'a':
                opt_abort = false;
                break;
        case 'A':
                opt_abort = true;
                break;
        case 'b':
#ifdef MALLOC_BALANCE
                opt_balance_threshold >>= 1;
#endif
                break;
        case 'B':
#ifdef MALLOC_BALANCE
                if (opt_balance_threshold == 0)
                        opt_balance_threshold = 1;
                else if ((opt_balance_threshold << 1)
                    > opt_balance_threshold)
                        opt_balance_threshold <<= 1;
#endif
                break;
```

```
#ifdef MALLOC_FILL
#ifndef MALLOC_PRODUCTION
        case 'c':
                opt_poison = false;
                break;
        case 'C':
                opt_poison = true;
                break;
#endif
#endif
        case 'f':
                opt_dirty_max >>= 1;
                break;
        case 'F':
                if (opt_dirty_max == 0)
                        opt_dirty_max = 1;
                else if ((opt_dirty_max << 1) != 0)
                        opt_dirty_max <<= 1;
                break;
#ifdef MALLOC_FILL
#ifndef MALLOC_PRODUCTION
        case 'j':
                opt_junk = false;
                break;
        case 'J':
                opt_junk = true;
                break;
#endif
#endif
#ifndef MALLOC_STATIC_SIZES
        case 'k':
                /*
                 * Chunks always require at least one
                 * header page, so chunks can never be
                 * smaller than two pages.
                 */
                if (opt_chunk_2pow > pagesize_2pow + 1)
                        opt_chunk_2pow--;
                break;
        case 'K':
                if (opt_chunk_2pow + 1 <
                    (sizeof(size_t) << 3))
                        opt_chunk_2pow++;
                break;
#endif
        case 'n':
                opt_narenas_lshift--;
                break;
        case 'N':
                opt_narenas_lshift++;
```

```
            break;
    /* More code ... */
```

4 Latently Unused Parameter

Derived from: Long Parameter List & Speculative Generality

Original description

> Long parameter lists are hard to understand, because they become inconsistent and difficult to use, and because you are forever changing them as you need more data. [. . .] Methods with unused parameters should be subject to Remove Parameter.

Variability-Aware Description: Some parameters in a method signature are only needed by certain features, but do not make sense for others. However, variable method signatures introduce another set of problems (Rosenmüller et al., GPCE 2007). Hence, all potential parameters should be forward-declared upon introduction of the method. However, if the feature that actually uses an optional parameter is absent, the method has unused parameters, which hinders program comprehension and may be likely to be a source of errors.

Applies to: Annotation-based and composition-based implementation techniques

Examples: Listing 1.3 shows an example for Latently Unused Parameter in the Graph Product Line, which is implemented in FOP.

The next listing (Listing 1.4) shows an artificial annotation-based example, which uses the C preprocessor.

Listing 1.3. Latently Unused Parameter in the Graph Product Line (GPL): `addAnEdge` Method from Class `Graph` in Features *DirectedOnlyVertices* and *WeightedOnlyVertices*

```
// Feature 'DirectedOnlyVertices' ('weight' is ignored)
public class Graph {
  /* more source code ... */

  public void addAnEdge( Vertex start,  Vertex end, int weight )
  {
    addEdge( start,end );
  }

  /* more source code ... */
}

// Feature 'WeightedOnlyVertices' ('weight' is used)
public class Graph {
  /* more source code ... */

  public void addAnEdge( Vertex start,  Vertex end, int weight )
  {
    addEdge( start,end, weight );
  }

  public void addEdge( Vertex start,  Vertex end, int weight )
  {
    addEdge( start,end );
    start.addWeight( weight );
    /* more source code ... */
  }
  /* more source code ... */
}
```

Listing 1.4. Latently Unused Parameter Using Annotations

```
class Stack {
  void push(Object elem, Transaction txn) {
#ifdef SYNC
    if (elem==null || txn==null) return;
    Lock l = txn.lock(elem);
#else
    if (elem==null) return;
#endif
    elementData[size++] = elem;
#ifdef SYNC
    l.unlock();
#endif
    fireStackChanged();
  }
}
```

5 Large Feature

Derived from: Large Class

Original description

> When a class is trying to do too much, it often shows up as too many instance variables. When a class has too many instance variables, duplicated code cannot be far behind. [. . .] As with a class with too many instance variables, a class with too much code is prime breeding ground for duplicated code, chaos, and death.

Variability-Aware Description: Like classes in single systems, features in a product line can also become *too* large. There may be different reasons such as mixing up concerns in a feature or a bad design of the SPL. However, at the end, the corresponding Large Feature is trying to do too much and thus, the whole design may be centered around this feature (similar to the *God Class* anti pattern).

Simple signs of this smell are large numbers of lines of code and the introduction of many new functions and / or classes (e.g., compared to all other features). Even if a consumer of the Large Feature only needs a portion of the feature's functionality, unwanted parts cannot be excluded. This hinders customization and may influence even non-functional properties such as footprint.

The remedy is simple: The Large Feature should be teased apart into several features; feature model and configurations have to be updated accordingly.

Applies to: Annotation-based and composition-based implementation techniques

Example: Due to the nature of the smell Large Feature, a code example is omitted.

6 Switch Statements with Optional Cases

Derived from: Switch Statements

Original description

> The problem with switch statements is essentially that of duplication. Often you find the same switch statement scattered across a program in different places. If you add a new clause to the switch, you have to find all these switch statements and change them. The object-oriented notion of polymorphism gives you an elegant way to deal with this problem.

Variability-Aware Description: A feature introduces additional `case` clauses in a `switch` statement. If the feature is not present, the `case` clause is inactive.

Applies to: Annotation-based and composition-based implementation techniques

Variability-Aware Description (for composition-based implementation techniques): With composition-based implementation techniques, such as FOP, introducing additional `case` clauses into a `switch` statement is not directly possible. However, this is sometimes emulated by combining a `switch` or `if` statement with calls to `original` or `Super`. Considering the example in Listing 1.5, these calls somewhat obfuscate the actual switch-case statement and thus, may impede its comprehension.

Variability-Aware Description (for annotation-based implementation techniques): Using annotation-based techniques, the optional `case` clauses can simply be guarded by a preprocessor directive. If the same `switch` occurs in multiple places, each occurence has to have the same guarded `case` clauses.

Examples: Listing 1.5 shows an example for a `switch` statement with optional `case` clauses in the FOP product line TankWar.

The next listing (Listing 1.6) shows an annotation-based example, taken from the Mozilla NSS library, file `lib/softoken/pkcs11.c`, which is part of Firefox version 28. It shows a `switch` statement with the optional `case CKK_EC`. In total, this `switch` is repeated in four different functions in the module. The listing only shows the first and second occurrence.

Listing 1.5. Example for a switch statement with optional cases in FOP
Adapted from the FeatureHouse-SPL TankWar:
`init` Method from Class `Tool` in (optional, sibling) Features *Bomb, Freeze, Firepower*

```
// Feature 'Bomb'
protected void init(TankManager manager, int xPos, int yPos,
    int toolType) {
  original(manager,xPos,yPos,toolType);
  switch (toolType) {
  case 374:
    original(manager, xPos * manager.grain2, yPos * manager.
        grain2, 255, 255, 0, manager.grain, manager.grain,
        toolType);
    break;
  }
}

// Feature 'Freeze'
protected void init(TankManager manager, int xPos, int yPos,
    int toolType) {
  original(manager, xPos, yPos, toolType);
  switch (toolType) {
  case 371:
    original(manager, xPos * manager.grain2, yPos * manager.
        grain2, 100, 149, 237, manager.grain, manager.grain,
        toolType);
    break;
  }
}

// Feature 'Firepower'
protected void init(TankManager manager, int xPos, int yPos,
    int toolType) {
  original(manager, xPos, yPos, toolType);
  switch (toolType) {
  case 372:
    original(manager, xPos * manager.grain2, yPos * manager.
        grain2, 0, 255, 0, manager.grain, manager.grain,
        toolType);
    break;
  }
}
```

Listing 1.6. Repeated Switch With Optional `case` Clause In Mozilla NSS Library, `security/nss/lib/softoken/pkcs11.c`

```
// 1st occurrence
static CK_RV
pk11_handlePublicKeyObject(PK11Session *session, PK11Object *
    object, CK_KEY_TYPE key_type)
{
  /* More code */
  switch (key_type) {
  /* More code */
  case CKK_DH:
    /* More code */
    break;
#ifdef NSS_ENABLE_ECC
  case CKK_EC:
    if ( !pk11_hasAttribute(object, CKA_EC_PARAMS)) {
      return CKR_TEMPLATE_INCOMPLETE;
    }
    if ( !pk11_hasAttribute(object, CKA_EC_POINT)) {
      return CKR_TEMPLATE_INCOMPLETE;
    }
    pubKeyAttr = CKA_EC_POINT;
    derive = CK_TRUE;     /* for ECDH */
    verify = CK_TRUE;     /* for ECDSA */
    encrypt = CK_FALSE;
    recover = CK_FALSE;
    wrap = CK_FALSE;
    break;
#endif /* NSS_ENABLE_ECC */
  default:
    /* More code */
}

// 2nd occurrence
static CK_RV
pk11_handlePrivateKeyObject(PK11Session *session,PK11Object *
    object,CK_KEY_TYPE key_type)
{
  /* More code */
  switch (key_type) {
    /* More code */
  case CKK_DH:
    /* More code */
    break;
#ifdef NSS_ENABLE_ECC
  case CKK_EC:
    if ( !pk11_hasAttribute(object, CKA_EC_PARAMS)) {
      return CKR_TEMPLATE_INCOMPLETE;
    }
    if ( !pk11_hasAttribute(object, CKA_VALUE)) {
      return CKR_TEMPLATE_INCOMPLETE;
    }
    if ( !pk11_hasAttribute(object, CKA_NETSCAPE_DB)) {
      return CKR_TEMPLATE_INCOMPLETE;
    }
    encrypt = CK_FALSE;
    recover = CK_FALSE;
    wrap = CK_FALSE;
    derive = CK_TRUE;
    break;
#endif /* NSS_ENABLE_ECC */
  default:
    /* More code */
}
```