Politecnico di Milano

Scuola di Ingegneria dell'Informazione

Corso di Laurea Magistrale di Computer Science and Engineering

Software Engineering 2 Project

Part III
Code Inspection

Principal Adviser:
Prof. Raffaela Mirandola

Authors:

Turconi Andrea ID n. 853589

Raimondi Lorenzo ID n. 859001

Accademic year 2015-2016

Contents

1 Classes

The class we were assigned to is

```
MethodAnnotater.java
```

placed in the package

```
package com.sun.jdo.api.persistence.enhancer.impl
```

More precisely, MethodAnnotater is a class contained in the CMP (Container Managed Persistance) module, in particular in the Enhancer component of Glassfish appserver. Container-Managed-Persistance is a method to manage persistent data within an entity bean; using CMP the persistence management is done without any implementation within the bean, beacuse the Container will invoke a Persistent Manager on behalf of the entity bean. In particular, MethodAnnotater class is used to handle and control code annotation of class methods involved with persistence, at Java VM instruction level. This class implements AnnotationConstants interface, a collection of costants defined in Java VM specification and useful for generating annotation. MethodAnnotater also extends Support class, which simply supports assertion and error signaling for the Enhancer component. The class also owns three internal classes: InsnNote, StackState and AnnotationFragment. InsnNote represents an instruction note, records some useful information related to a java instruction. StackState represents an association between a certain instruction and its operands' depth on the stack. AnnotationFragment simply associate an instruction with the required words of stack used for its execution.

2 Functional Role

Here are presented the functional role explaination of the methods assigned to us. We tried to understand what are the aims of the class and of this single methods by searching and analyzing the source code of MethodAnnotater and of relative classes used by it or contained in the same package. We also used Glassfish and Java documentations, beside with JVM Specification.

2.1 buildBasicAnnotation (IsnsNote note)

```
buildBasicAnnotation(InsnNote note)
```

This method generates a Java VM Instruction sequence starting from the InsnNote object related to the target java instruction to process. The methods first gets from the InsnNote target class name, field name and root class, and than generates an initial instruction sequence. After this standard sequence creation, new VM instruction are appended to it on the base of the previously got instruction attributes, regarding JDOFlags and StateManager. Once created the method returns an AnnotationFragment reporting the generated instruction sequence and the word of stack required for its execution.

2.2 findArgdepositer (Insn currInsn, int argDepth)

```
findArgDepositer(Insn currInsn, int argDepth)
```

This function is in charge of locating the instruction that is argDepth deep in the stack and which deposits 1-word stack argument to current instruction (top of the stack == 0). If the instruction is a Target instruction, the search will be aborted for this operation and the search continues for the following instruction. Don't return depositors different from a duplication operation that return more than one word.

2.3 minimizeStack (StackState state)

```
minimizeStack(StackState state)
```

This function is in charge of search an operation that allocate the minimum depth word at the top of the stack. This function aborts the search for a target operation. If there isn't a smaller operation, this function doesn't change the stack state. If there is a swap operation, this function computes the swapping before evaluate how long are the words swapped, otherwise calculates the length of operation's arguments.

3 List of Issues

3.1 buildBasicAnnotation (InsnNote note)

Naming Conventions

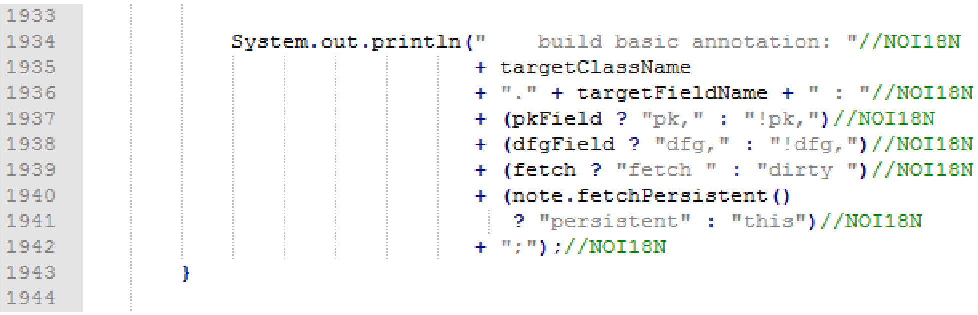
1. *All class names, interface names, method names, class variables, method variables, and constants used should have meaningful names and do what the name suggests.*

   The class Insn name is not so meaningful. Insn stands for Instruction Sequence, maybe the extended name or another abbreviation would have been better.

Wrapping Lines

1. *Line break occurs after a comma or an operator.*

   Lines 1934-1939, 1941, 1942, 1990, 1991: line break occurs before the "+" operator, and not after it.

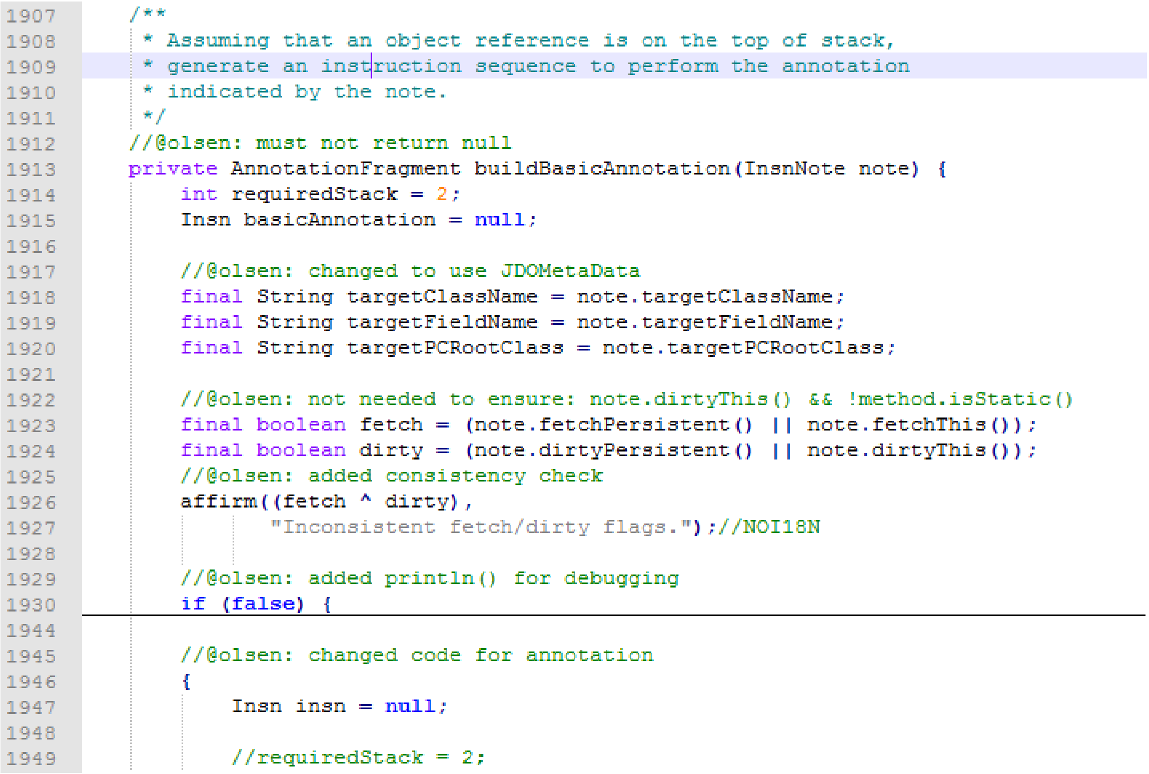


```
1933
1934            System.out.println("    build basic annotation: "//NOI18N
1935                                + targetClassName
1936                                + "." + targetFieldName + " : "//NOI18N
1937                                + (pkField ? "pk," : "!pk,")//NOI18N
1938                                + (dfgField ? "dfg," : "!dfg,")//NOI18N
1939                                + (fetch ? "fetch " : "dirty ")//NOI18N
1940                                + (note.fetchPersistent()
1941                                  ? "persistent" : "this")//NOI18N
1942                                + ";");//NOI18N
1943        }
1944
```

Comments

1. *Commented out code contains a reason for being commented out and a date it can be removed from the source file if determined it is no longer needed.*

   Line 1949: commented instruction without any reason and any remove date. The instruction is also redundant because the same assignment is already present in line 1914, and between this two occurrence the variable is never modified or used.



```
/**
 * Assuming that an object reference is on the top of stack,
 * generate an instruction sequence to perform the annotation
 * indicated by the note.
 */
//@olsen: must not return null
private AnnotationFragment buildBasicAnnotation(InsnNote note) {
    int requiredStack = 2;
    Insn basicAnnotation = null;

    //@olsen: changed to use JDOMetaData
    final String targetClassName = note.targetClassName;
    final String targetFieldName = note.targetFieldName;
    final String targetPCRootClass = note.targetPCRootClass;

    //@olsen: not needed to ensure: note.dirtyThis() && !method.isStatic()
    final boolean fetch = (note.fetchPersistent() || note.fetchThis());
    final boolean dirty = (note.dirtyPersistent() || note.dirtyThis());
    //@olsen: added consistency check
    affirm((fetch ^ dirty),
           "Inconsistent fetch/dirty flags.");//NOI18N

    //@olsen: added println() for debugging
    if (false) {

    //@olsen: changed code for annotation
    {
        Insn insn = null;

        //requiredStack = 2;
```

3.2 findArgdepositer (Insn currInsn, int argDepth)

Naming Conventions

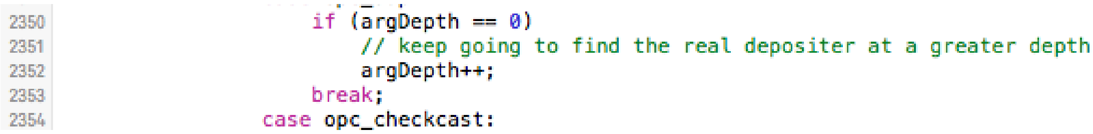
1. *If one-character variables are used, they are used only for temporary "throw-away" variables, such as those used in for loops.*

   One-character variable 'i' is not used as 'throwaway' variable, but it is used as a standard variable.

Braces

1. *All if, while, do-while, try-catch, and for statements that have only one statement to execute are surrounded by curly braces.*

   At line 2352 'if' statements that have only one statement to execute is not surrounded by curly braces.

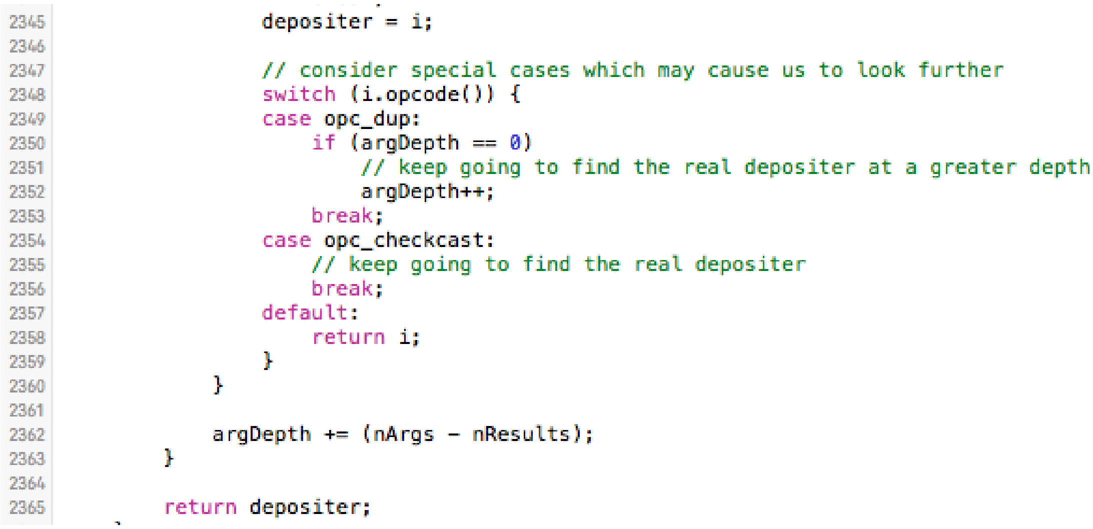


```
2350                 if (argDepth == 0)
2351                     // keep going to find the real depositer at a greater depth
2352                     argDepth++;
2353                 break;
2354             case opc_checkcast:
```

File Organization

1. *Blank lines and optional comments are used to separate sections (beginning comments, package/import statements, class/interface declarations which include class variable/attributes declarations, constructors, and methods).*

   At lines 2346, 2361, 2364 there are blank lines used to separate code that doesn't belong to different sections.

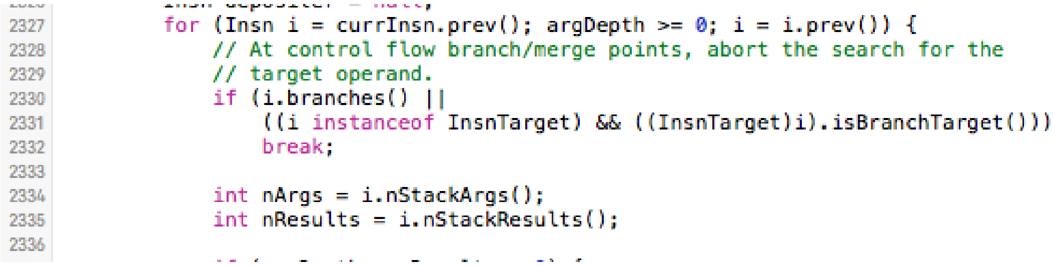


```
2345                depositer = i;
2346
2347                // consider special cases which may cause us to look further
2348                switch (i.opcode()) {
2349                case opc_dup:
2350                    if (argDepth == 0)
2351                        // keep going to find the real depositer at a greater depth
2352                        argDepth++;
2353                    break;
2354                case opc_checkcast:
2355                    // keep going to find the real depositer
2356                    break;
2357                default:
2358                    return i;
2359                }
2360            }
2361
2362            argDepth += (nArgs - nResults);
2363        }
2364
2365        return depositer;
```

Initialization and Declarations

1. *Declarations appear at the beginning of blocks (A block is any code surrounded by curly braces "{" and "}"). The exception is a variable can be declared in a 'for' loop.*

   Declarations of variables at lines 2334,2335, are not the first statements of the related blocks



```
2327        for (Insn i = currInsn.prev(); argDepth >= 0; i = i.prev()) {
2328            // At control flow branch/merge points, abort the search for the
2329            // target operand.
2330            if (i.branches() ||
2331                ((i instanceof InsnTarget) && ((InsnTarget)i).isBranchTarget()))
2332                break;
2333
2334            int nArgs = i.nStackArgs();
2335            int nResults = i.nStackResults();
2336
```

3.3 minimizeStack (StackState state)

Naming Conventions

1. *All class names, interface names, method names, class variables, method variables, and constants used should have meaningful names and do what the name suggests.*

   "Object x,y,o" respectively at lines 2410, 2411, 2453 have not meaningful names and also those are one-character name variables used not as 'throwaway variables.

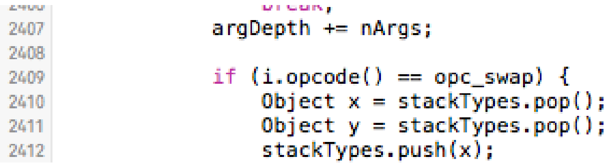
2. *If one-character variables are used, they are used only for temporary "throwaway" variables, such as those used in for loops.*

   One-character variable 'i' is not used as 'throwaway' variable, but it is used as a standard variable.

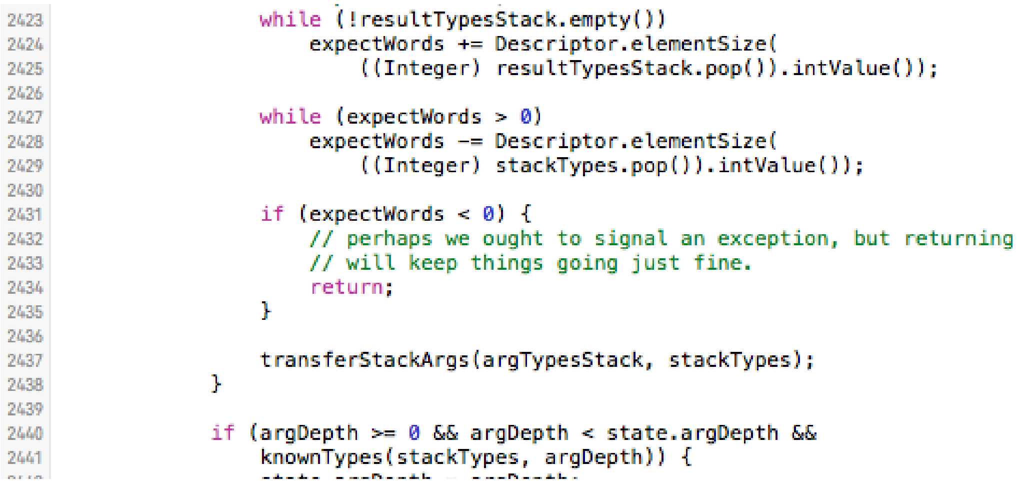
File Organization

1. *Blank lines and optional comments are used to separate sections (beginning comments, package/import statements, class/interface declarations which include class variable/attributes declarations, constructors, and methods).*

   At lines 2408, 2426, 2430, 2436, 2439 there are blank lines used to separate code that doesn't belong to different sections.



```
2406                    break;
2407                argDepth += nArgs;
2408
2409                if (i.opcode() == opc_swap) {
2410                    Object x = stackTypes.pop();
2411                    Object y = stackTypes.pop();
2412                    stackTypes.push(x);
```

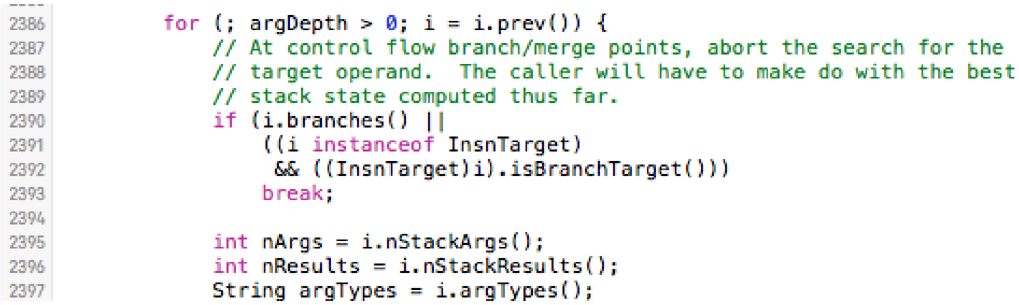


```
2423                while (!resultTypesStack.empty())
2424                    expectWords += Descriptor.elementSize(
2425                        ((Integer) resultTypesStack.pop()).intValue());
2426
2427                while (expectWords > 0)
2428                    expectWords -= Descriptor.elementSize(
2429                        ((Integer) stackTypes.pop()).intValue());
2430
2431                if (expectWords < 0) {
2432                    // perhaps we ought to signal an exception, but returning
2433                    // will keep things going just fine.
2434                    return;
2435                }
2436
2437                transferStackArgs(argTypesStack, stackTypes);
2438            }
2439
2440            if (argDepth >= 0 && argDepth < state.argDepth &&
2441                knownTypes(stackTypes, argDepth)) {
2442                state.argDepth = argDepth;
```

Initialization and Declarations

1. *Declarations appear at the beginning of blocks (A block is any code surrounded by curly braces "{" and "}"). The exception is a variable can be declared in a 'for' loop.*

   Declarations of variables at lines 2395,2396 are not the first statements of the related blocks

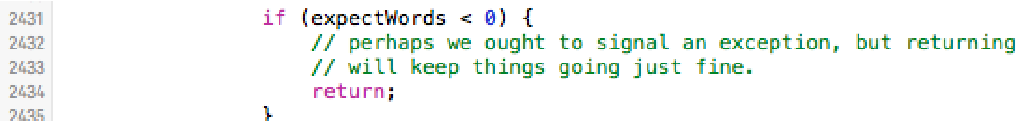


```
2386        for (; argDepth > 0; i = i.prev()) {
2387            // At control flow branch/merge points, abort the search for the
2388            // target operand.  The caller will have to make do with the best
2389            // stack state computed thus far.
2390            if (i.branches() ||
2391                ((i instanceof InsnTarget)
2392                 && ((InsnTarget)i).isBranchTarget()))
2393                break;
2394
2395            int nArgs = i.nStackArgs();
2396            int nResults = i.nStackResults();
2397            String argTypes = i.argTypes();
```

Exceptions

1. *Check that the relevant exceptions are caught*

   At line 2431 it is used an 'if' control to catch exception, but this is not the right way to solve this problem

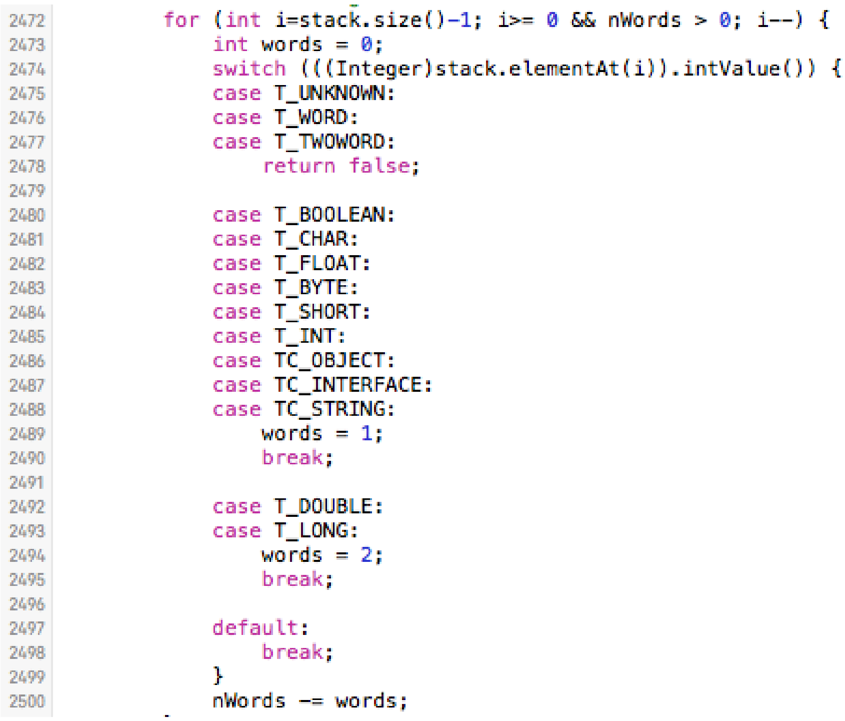


```
2431                if (expectWords < 0) {
2432                    // perhaps we ought to signal an exception, but returning
2433                    // will keep things going just fine.
2434                    return;
2435                }
```

Flow of Control

1. *In a switch statement, check that all cases are addressed by break or return*

   At lines 2475, 2476, 2480, 2481, 2482, 2483, 2484, 2485, 2486, 2487, 2492 of a 'switch' sequence, there are not the 'break' statement



```
2472    for (int i=stack.size()-1; i>= 0 && nWords > 0; i--) {
2473        int words = 0;
2474        switch (((Integer)stack.elementAt(i)).intValue()) {
2475        case T_UNKNOWN:
2476        case T_WORD:
2477        case T_TWOWORD:
2478            return false;
2479
2480        case T_BOOLEAN:
2481        case T_CHAR:
2482        case T_FLOAT:
2483        case T_BYTE:
2484        case T_SHORT:
2485        case T_INT:
2486        case TC_OBJECT:
2487        case TC_INTERFACE:
2488        case TC_STRING:
2489            words = 1;
2490            break;
2491
2492        case T_DOUBLE:
2493        case T_LONG:
2494            words = 2;
2495            break;
2496
2497        default:
2498            break;
2499        }
2500        nWords -= words;
```

4 Other problems

buildBasicAnnotation method contains two unreachable code blocks used for printing to console some debugging information. Since unreachable code is not a good programming practice its use should be avoided. A simple solution to this problem may be introducing a boolean debug variable to avoid/enter these blocks, whose value should be passed as method argument by the caller.

For what concerns commented out code, across all the MethodAnnotater class there are a lot of commented out instructions, methods and nested classes without any reasonable explaination or final removing date. We counted 1130 lines of commented out code: these, on the 2849 total class lines, imply a 40% of commented code on the overall file, that is a considerably high amount.