Università della Svizzera Italiana

Knowledge Analysis & Management

Bad Smells

Simone Masiero simone.masiero@usi.ch

October 23, 2020

Overview

The Goal of this project is to exploit ontologies as approach find bad smells in code like Bloaters, Object-Orientation Abusers, change preventers, dispensables and couplers. Even if the structure of this project allows to analyze any java project, we will use, as subject of our experiment, AndroidChess, an app to play chess on Android smartphone or tablet.

Structure

This project is entirely done in Python; the whole code is reported in the appendix and the project file could be interactively run with Jupiter notebook. Moreover, the Github repository for this project could be found here.

1 Ontology creation

1.1 Process

In this first phase of the project, our goal is to encode the Java language as an ontology, hence, for each Java entities (Class, Method, Field, Statement, etc.) we will have a corresponding class in our ontology whit all the required nesting. In order to accomplish this task, we will exploit the structure of the file `tree.py` from JavaLang, a pure Python library which provides a lexer and parser targeting Java 8. This file provides us with a tree representation of all the Java entities, parsing it, we could construct our ontology.

In order to achieve this, firstly we need to import some basic libraries like `path`, to access files and `ast` to parse abstract syntax trees, then we will need `owlready2`, a module for ontology-oriented programming in Python. Then, we will declare, for convenience, four string which will hold some relative path in our project.

(code for this step: A.1)

Once done with the importing phase, we could start creating our java ontology. Firstly we will instantiate a new ontology through the command `get_ontology("http://Java_Ontology/JavaTree.owl")`, after, we need to dynamically add to the ontology each java entity while parsing the `tree.py` file. We tackled this problem by extending the `ast.NodeVisitor` class with our `Visitor` class (which inherits from `ast.NodeVisitor`) and redefines the parent's function `generic_visit` to traverse the ast generated from `tree.py` adding the code needed to save each visited entity to our ontology.

Each java entity in `tree.py` is encoded as a class, if it inherits from `Node` we know it's a base java entity and we have to extend this from `Things` in our ontology, else, if it inherits from another java entity, we have to specify the necessaries dependency in our ontology (e.g., the entity `PackageDeclaration` inherits from `Declaration` and `Documented`, so in our ontology it will have a `SubClassOf` relation with those two)

Lastly, we have to pay attention to the attributes of each class, which we have to encode as `ObjectProperty` or `DataProperty` in our ontology.

(code for this step: A.2; **NB**: for datatype properties we renamed property "name" to "jname" to avoid conflicts with the predefined "name" attribute of ontology instances, moreover, we assumed all property to be data properties, except for "body" and "parameters", which are Object properties, as stated in the assignment.)

Once our visit finishes, the ontology is now representing the generic java language. We will save this in a file called tree.owl under the `/computed` folder in rdfxml format.

(code for this step: A.3)

1.2 Structure Stats

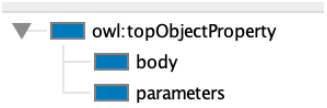

Figure 1: Structure of object properties in our ontology

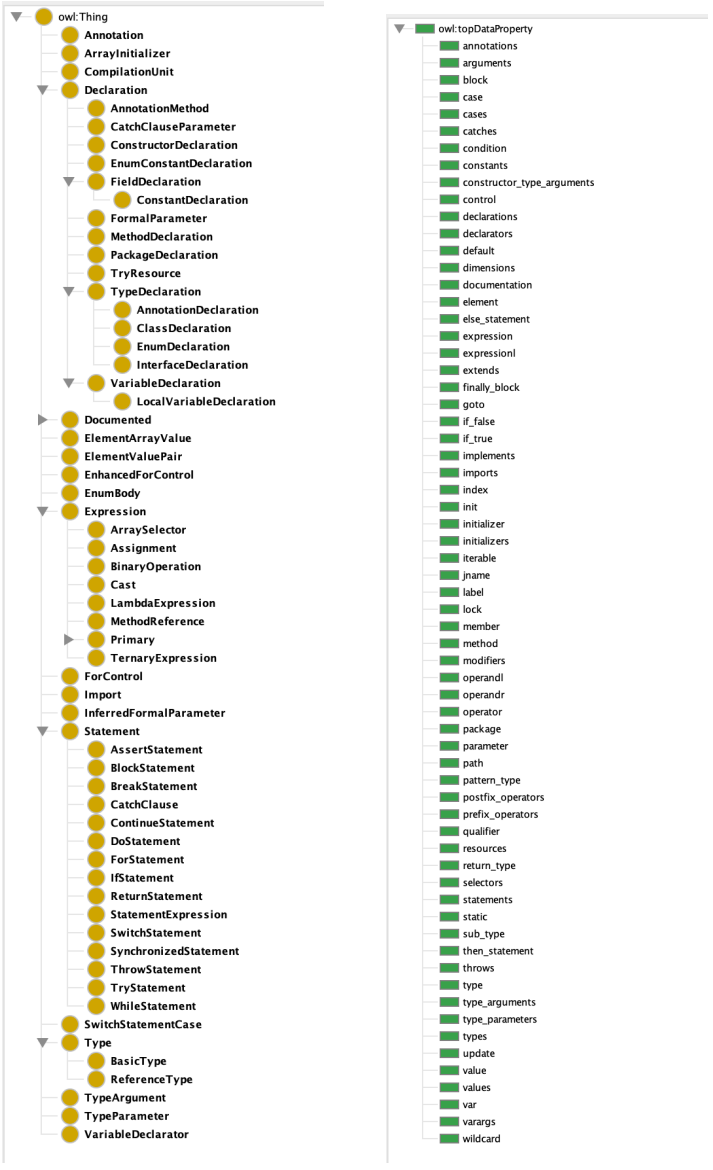

Figure 2: Structure of Classes (left) and Data Types (right) in our ontology

As we can see from figures 1 and 2 we have 78 classes in our ontology, 65 data properties and 2 object properties

1.3 Tests

Unit test from this step are all following the same schema (one example is reported in A.4). We load our java ontology and we access one instance, for example `ClassDeclaration`, then we just check that all names, and relations are correct.

2 Creation of ontology instances

2.1 Process

In this second step, we will populate our java ontology based on the content of the classes found on our project, hence, for each class we will store the entities encountered with proper relations and hierarchy positions.

In order to do this, we will firstly extract all the classes in the project folder we want to analyze (`./resources/android-chess/` without considering sub-directories). Each java class is parsed and saved in a list so that we have everything in one place ready for the population step. After that we will retrieve our ontology and, looping through the previously generated ASTs of each java class, for each node in the tree we will find out if its a field, a method or a constructor and we create the proper instance in the ontology accordingly. **N.B.** in case it is a method or a constructor, we will also create in the ontology the paramenters and statements associated with it. (code for this step: B.1)

Finally we save our populated ontology in a new rdfxml file called `tree2.owl` under the `computed` folder (code for this step: B.2)

2.2 Stats

Class	# of individuals
ClassDeclaration	11
MethodDeclaration	152
FieldDeclaration	105
ConstructorDeclaratio	6
FormalParameter	165
AssertStatement	0
BlockStatement	143
BreakStatement	23
CatchClause	8
ContinueStatement	4
DoStatement	2
ForStatement	6
IfStatement	125
ReturnStatement	106
StatementExpression	446
SwitchStatement	8
SynchronizedStatemen	1
ThrowStatement	15
TryStatement	8
WhileStatement	10

Table 1: Number of individuals for each class

2.3 Tests

As for unit tests in this section, what we do is parsing a custom java class, we introduced some example class and add that to the ontology in the same way we added classes for the project. Then we assert that specific fields, methods, constructors, parameters and statements are save into the ontology in the proper way; we reported an example in the appendix. (code for this step: B.3)

3 Bad smell detection

3.1 Logic

The logic for this step is pretty straightforward, we just model the queries as Sparql queries and we run them on the graph representing our ontology; Then we save the output of each query in their related text files. (code for this step: C.1 and C.2)

3.2 Stats

Bad Smell	# of smells found
Data classes	1
Constructors with long parameter list	0
Methods with long parameter list	4
Constructors with switch statements	0
Method with switch statements	8
large classes	3
Long constructors	0
Long methods	10

Table 2: Number of bad smells found for each category

3.2.1 Data classes

Class	# Getter/Setters
Valuation	1

Table 3: Data classes found, classes with only getters and setters

3.2.2 Methods with long parameter list

Class	Method	# of parameters
PGNProvider	query	5
ChessPuzzleProvider	query	5
GameControl	addPGNEntry	5
JNI	setCastlingsEPAnd50	6

Table 4: Methods with long parameter list found, more than 4

3.2.3 Methods with switch statements

Class	Method	# of switch statements
PGNProvider	query	1
PGNProvider	getType	1
PGNProvider	delete	1
PGNProvider	update	1
ChessPuzzleProvider	query	1
ChessPuzzleProvider	getType	1
ChessPuzzleProvider	delete	1
ChessPuzzleProvider	update	1

Table 5: Method with switch statements found

3.2.4 Large Classes

Class	# of methods
GameControl	63
JNI	44
Move	21

Table 6: Large Classes found, with more than 10 methods

3.2.5 Long methods

Class	Method	# of statements
PGNProvider	insert	31
ChessPuzzleProvider	query	25
ChessPuzzleProvider	insert	20
GameControl	loadPGNHead	26
GameControl	loadPGNMoves	96
GameControl	requestMove	76
GameControl	getDate	26
JNI	newGame	35
JNI	initFEN	88
JNI	initRandomFisher	87

Table 7: Long methods found, with more than 19 statements

3.3 Tests

Unit tests for this step are similar to the ones done for step two, except that this time we create java programs which are specifically manifesting a bad smell, then we try assert that bad smell with out queries. One example of unit test can be found in C.3

A Step 1

A.1 Imports

```
import sys

sys.path.append('/usr/local/lib/python3.8/site-packages')

from os import path as Path
import ast
from owlready2 import *
import javalang.tree
import rdflib
import rdflib.plugins.sparql as sq

# relative path of the subject code folder
chess_path              = './resources/android-chess/app/src/main/java/jwtc/chess/'
# relative path of the java tree structure
treePy_path             = './resources/tree.py'
# relative path of our populated ontology
populated_ontology_path = './computed/tree2.owl'
# relative path of our empty ontology
ontology_path           = './computed/tree.owl'
# relative path for queries
queries_path           = './queries/'
```

A.2 Visitor Class

```
class Visitor(ast.NodeVisitor):

    def __init__(self, ontology):
        self.ontology = ontology

    def generic_visit(self, node):
        ast.NodeVisitor.generic_visit(self, node)
        with self.ontology as onto:
            if type(node) == ast.ClassDef:
                for obj in node.bases:
                    if obj.id == "Node":
                        types.new_class(node.name, (Thing,))
                    else:
                        types.new_class(node.name, (onto[obj.id],))

            elif type(node) == ast.Assign:
                for el in node.value.elts:
                    if el.s == "body" or el.s == "parameters":
                        types.new_class(el.s, (ObjectProperty,))
                    else:
                        types.new_class(
                            "jname" if el.s == "name" else el.s,
                            (DataProperty,))
```

A.3 Ontology creation

```
def create_ontology(path):
    with open(path, "r") as source:
        tree = ast.parse(source.read())
```

```
4
5       ontology = get_ontology("http://Java_Ontology/JavaTree.owl")
6
7       visitor = Visitor(ontology)
8       visitor.visit(tree)
9       if os.path.exists(ontology_path):
10          os.remove(ontology_path)
11      ontology.save(ontology_path, format="rdfxml")
12
13  create_ontology("./resources/tree.py")
```

A.4 Unit Tests 1

```
1   onto = get_ontology(ontology_path).load()
2   cd = onto["ClassDeclaration"]
3
4   assert cd.name == "ClassDeclaration"
5   assert len(cd.is_a) == 1
6   assert cd.is_a[0].name == 'TypeDeclaration'
```

B Step 2

B.1 Helper functions

```
1   def append_fields(class_declaration, fields, ontology):
2       for field in fields:
3           o_fd = ontology['FieldDeclaration']()
4           o_fd.jname = [field.name]
5           class_declaration.body.append(o_fd)
6
7   def append_method(class_declaration, method, ontology):
8       o_md = ontology['MethodDeclaration']()
9       o_md.jname = [method.name]
10      append_statements(o_md, method, ontology)
11      append_parameters(o_md, method, ontology)
12      class_declaration.body.append(o_md)
13
14  def append_constructor(class_declaration, method, ontology):
15      o_con = ontology['ConstructorDeclaration']()
16      o_con.jname = [method.name]
17      append_statements(o_con, method, ontology)
18      append_parameters(o_con, method, ontology)
19      class_declaration.body.append(o_con)
20
21  def append_statements(md, method, ontology):
22      for _, statement in method.filter(javalang.tree.Statement):
23          if type(statement) != javalang.tree.Statement:
24              s_type = statement.__class__.__name__
25              s = ontology[s_type]()
26              md.body.append(s)
27
28  def append_parameters(md, method, ontology):
29      for _, statement in method.parameters:
30          fp = ontology['FormalParameter']()
31          md.parameters.append(fp)
```

B.2 Main Logic

```
def extract_class_declaration(source):
    class_declarations = []
    for file in os.listdir(source):
        if file.endswith('.java'):
            f_path = os.path.join(source, file)
            with open(f_path) as j_file:
                ast = javalang.parse.parse(j_file.read())
                for _, node in ast.filter(javalang.tree.ClassDeclaration):
                    class_declarations.append(node)
    return class_declarations

def addClasses(ontology, class_declarations):
    for java_class in class_declarations:
        ontology_cd = ontology['ClassDeclaration']()
        ontology_cd.jname = [java_class.name]
        [append_fields(ontology_cd, f.declarators, ontology) for f in java_class.body if type(f) == javalang.tree.FieldDeclarati
        [append_method(ontology_cd, m, ontology) for m in java_class.body if type(m) == javalang.tree.MethodDeclaration]
        [append_constructor(ontology_cd, c, ontology) for c in java_class.body if type(c) == javalang.tree.ConstructorDeclaratio

def add_instances(ontology_path, source_path):
    class_declarations = extract_class_declaration(source_path)
    ontology = get_ontology(ontology_path).load()
    with ontology:
        addClasses(ontology, class_declarations)
    if os.path.exists(populated_ontology_path):
        os.remove(populated_ontology_path)
    ontology.save(populated_ontology_path, format="rdfxml")

add_instances(ontology_path, chess_path)
```

B.3 Test

```
onto = get_ontology(ontology_path).load()
tree = javalang.parse.parse('class Main { int sum; Main() { this(5, 2); } Main(int arg1, int arg2) { this.sum = arg1 + arg2; } v
cds = []
for _, node in tree.filter(javalang.tree.ClassDeclaration):
    cds.append(node)

with onto:
    addClasses(onto, cds)
if os.path.exists(populated_ontology_path):
    os.remove(populated_ontology_path)
onto.save(populated_ontology_path, format="rdfxml")

a = onto['ClassDeclaration'].instances()[-1]

assert a.jname[0] == 'Main'
assert a.body[0].is_a[0].name == 'FieldDeclaration'
assert a.body[0].jname[0] == 'sum'
assert a.body[1].is_a[0].name == 'MethodDeclaration'
assert a.body[1].jname[0] == 'display'
assert a.body[2].is_a[0].name == 'MethodDeclaration'
assert a.body[2].jname[0] == 'main'
assert a.body[3].is_a[0].name == 'ConstructorDeclaration'
assert a.body[3].jname[0] == 'Main'
assert a.body[4].is_a[0].name == 'ConstructorDeclaration'
assert a.body[4].jname[0] == 'Main'

```

```
assert a.body[4].parameters[0].is_a[0].name == 'FormalParameter'
assert a.body[4].parameters[1].is_a[0].name == 'FormalParameter'
```

C Step 3

C.1 Logic

```
def log(header, rows, logFunc):
    out = open(queries_path + "/" + header + ".txt", "w")
    for row in rows:
        out.write(logFunc(row))
    out.close()

if not os.path.exists(queries_path):
        os.makedirs(queries_path)

graph = default_world.as_rdflib_graph()
```

C.2 Queries

```
longMethods = sq.prepareQuery(
 """SELECT ?className ?methodName ?statements (COUNT(*)AS ?tot) WHERE {
                ?class a tree:ClassDeclaration .
                ?class tree:jname ?className .
                ?class tree:body ?method .
                ?method a tree:MethodDeclaration .
                ?method tree:jname ?methodName .
                ?method tree:body ?statement .
                ?statement a/rdfs:subClassOf* tree:Statement .
            } GROUP BY ?method
            HAVING (COUNT(?statement) >= 20)
        """,
 initNs = { "tree": "http://Java_Ontology/JavaTree.owl#" })

def longMethodsLog(row):
    return "CLASS: " + row.className + "\t METHOD: " + row.methodName + "\t STATEMENTS COUNT: " + row.tot + "\n"

res = graph.query(longMethods)
log("LongMethods", res, longMethodsLog)



longConstructors = sq.prepareQuery(
 """SELECT ?className ?constructorName ?statements (COUNT(*)AS ?tot) WHERE {
                ?class a tree:ClassDeclaration .
                ?class tree:jname ?className .
                ?class tree:body ?constructor .
                ?constructor a tree:ConstructorDeclaration .
                ?constructor tree:jname ?constructorName .
                ?constructor tree:body ?statement .
                ?statement a/rdfs:subClassOf* tree:Statement .
            } GROUP BY ?constructor
            HAVING (COUNT(?statement) >= 20)
        """,
 initNs = { "tree": "http://Java_Ontology/JavaTree.owl#" })

def longConstructorsLog(row):
```

```
38      return "CLASS: " + row.className + "\t CONSTRUCTOR: " + row.methodName + "\t STATEMENTS COUNT: " + row.tot + "\n"
39
40  res = graph.query(longConstructors)
41  log("LongConstructors", res, longConstructorsLog)
42
43
44
45  largeClasses = sq.prepareQuery(
46   """SELECT ?className ?methods (COUNT(*)AS ?tot) WHERE {
47                  ?class a tree:ClassDeclaration .
48                  ?class tree:jname ?className .
49                  ?class tree:body ?method .
50                  ?method a tree:MethodDeclaration .
51                  ?method tree:jname ?methodName .
52              } GROUP BY ?className
53              HAVING (COUNT(?method) >= 10)
54          """,
55   initNs = { "tree": "http://Java_Ontology/JavaTree.owl#" })
56
57  def largeClassesLog(row):
58      return "CLASS: " + row.className + "\t METHODS COUNT: " + row.tot + "\n"
59
60  res = graph.query(largeClasses)
61  log("LargeClasses", res, largeClassesLog)
62
63
64
65  methodWithSwitch = sq.prepareQuery(
66   """SELECT ?className ?methodName (COUNT(*)AS ?tot) WHERE {
67                  ?class a tree:ClassDeclaration .
68                  ?class tree:jname ?className .
69                  ?class tree:body ?method .
70                  ?method a tree:MethodDeclaration .
71                  ?method tree:jname ?methodName .
72                  ?method tree:body ?statement .
73                  ?statement a tree:SwitchStatement .
74              } GROUP BY ?method
75              HAVING (COUNT(?method) >= 1)
76          """,
77   initNs = { "tree": "http://Java_Ontology/JavaTree.owl#" })
78
79  def methodWithSwitchLog(row):
80      return "CLASS: " + row.className + "\t METHOD: " + row.methodName + "\t SWITCH COUNT: " + row.tot + "\n"
81
82  res = graph.query(methodWithSwitch)
83  log("MethodsWithSwitch", res, methodWithSwitchLog)
84
85
86
87  constructorWithSwitch = sq.prepareQuery(
88   """SELECT ?className ?constructorName (COUNT(*)AS ?tot) WHERE {
89                  ?class a tree:ClassDeclaration .
90                  ?class tree:jname ?className .
91                  ?class tree:body ?constructor .
92                  ?constructor a tree:ConstructorDeclaration .
93                  ?constructor tree:jname ?constructorName .
94                  ?constructor tree:body ?statement .
95                  ?statement a tree:SwitchStatement .
96              } GROUP BY ?constructor
97              HAVING (COUNT(?constructor) >= 1)
98          """,
99   initNs = { "tree": "http://Java_Ontology/JavaTree.owl#" })
100
```

```
def constructorWithSwitchLog(row):
    return "CLASS: " + row.className + "\t CONSTRUCTOR: " + row.methodName + "\t SWITCH COUNT: " + row.tot + "\n"

res = graph.query(constructorWithSwitch)
log("ConstructorWithSwitch", res, constructorWithSwitchLog)



methodWithLongParameterList = sq.prepareQuery(
 """SELECT ?className ?methodName ?parameter (COUNT(*)AS ?tot) WHERE {
                ?class a tree:ClassDeclaration .
                ?class tree:jname ?className .
                ?class tree:body ?method .
                ?method a tree:MethodDeclaration .
                ?method tree:jname ?methodName .
                ?method tree:parameters ?parameter .
                ?parameter a tree:FormalParameter .
            } GROUP BY ?method
            HAVING (COUNT(?parameter) >= 5)
        """,
 initNs = { "tree": "http://Java_Ontology/JavaTree.owl#" })

def MethodWithLongParameterListLog(row):
    return "CLASS: " + row.className + "\t METHOD: " + row.methodName + "\t PARAMETERS COUNT: " + row.tot + "\n"

res = graph.query(methodWithLongParameterList)
log("MethodWithLongParameterList", res, MethodWithLongParameterListLog)



contructorWithLongParameterList = sq.prepareQuery(
 """SELECT ?className ?constructorName ?parameter (COUNT(*)AS ?tot) WHERE {
                ?class a tree:ClassDeclaration .
                ?class tree:jname ?className .
                ?class tree:body ?constructor .
                ?constructor a tree:ConstructorDeclaration .
                ?constructor tree:jname ?constructorName .
                ?constructor tree:parameters ?parameter .
                ?parameter a tree:FormalParameter .
            } GROUP BY ?constructor
            HAVING (COUNT(?parameter) >= 5)
        """,
 initNs = { "tree": "http://Java_Ontology/JavaTree.owl#" })

def contructorWithLongParameterListLog(row):
    return "CLASS: " + row.className + "\t CONSTRUCTOR: " + row.constructorName + "\t PARAMETERS COUNT: " + row.tot + "\n"

res = graph.query(contructorWithLongParameterList)
log("ConstructorWithLongParameterList", res, contructorWithLongParameterListLog)



dataClass = sq.prepareQuery(
 """SELECT ?className (COUNT(*)AS ?tot) WHERE {
                ?class a tree:ClassDeclaration .
                ?class tree:jname ?className .
                ?class tree:body ?method .
                ?method a tree:MethodDeclaration .
                ?method tree:jname ?methodName .
                FILTER (regex(?methodName, "set.*") || regex(?methodName, "get.*"))
            } GROUP BY ?className
        """,
 initNs = { "tree": "http://Java_Ontology/JavaTree.owl#" })
```

```

retriveAllMethods = sq.prepareQuery(
    """SELECT ?className (COUNT(*) AS ?tot) WHERE {
                ?class a tree:ClassDeclaration .
                ?class tree:jname ?className .
                ?class tree:body ?method .
                ?method a tree:MethodDeclaration .
                ?method tree:jname ?methodName .
            } GROUP BY ?className
        """,
 initNs = { "tree": "http://Java_Ontology/JavaTree.owl#" })

resGS = graph.query(dataClass)
resAll = graph.query(retriveAllMethods)

out = open(queries_path + "/DataClasses.txt", "w")
for rowGS in resGS:
    for rowA in resAll:
        if rowGS.className == rowA.className:
            if rowGS.tot == rowA.tot:
                out.write("CLASS: " + rowGS.className + "\t GETTERS/SETTERS: " + rowGS.tot + "\n")
out.close()
```

C.3 Tests

```
onto = get_ontology(ontology_path).load()
tree = javalang.parse.parse("class A { int f(int x) { x++;x++;x++;x++;x++;x++;x++;x++;x++;x++;x++;x++;x++;x++;x++;x++;x++;x++;x+
cds = []
for _, node in tree.filter(javalang.tree.ClassDeclaration):
    cds.append(node)

with onto:
    addClasses(onto, cds)
if os.path.exists(populated_ontology_path):
    os.remove(populated_ontology_path)

onto.save(file="./test/tmp.owl", format="rdfxml")

g = rdflib.Graph()
g.load("./test/tmp.owl")
res = g.query(longMethods)
assert len(res) == 1
```