Assignment Report
Project: Bad smell detection
Simone Giacomelli

In this project, we use ontologies to find bad smells in code. The subject of this project is AndroidChess, an app to play chess on Android smart-phone or tablets.

Table of contents

Create ontology

In this section, I will develop a python source file ‘onto-creator.py’ that:

- accepts in input ‘tree.py’ from javalang package
- produce as output the ‘tree.owl’ file

The owl output file contains an ontology for Java entities:

- Class
- Method

- Field
- Statements
- … and so on.

Ontology creation code

We can divide the source code for this step (onto-creator.py) in three different parts:

- parsing of the Java entities
- generation of the ontology output file
- visit of the AST of the parsed source code to fill the output

The integrated python module ‘ast’ make the parsing of python source code very easy. It is just two lines of code:

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

The following step creates the empty ontology:

```
onto = get_ontology("http://usi.ch/giacomelli/Knowledge_project1.owl")
```

As we can see in the code above, we are requested to specify a namespace that will be reused later in the query phase.

The visit of the AST is conducted with the ‘visit’ pattern; these are the ingredients needed to implement such a pattern:

- a subclass of ast.NodeVisitor
- the implementation of visit_ClassDef for handling the visit to this entity

Inside the ‘visit_ClassDef(...)’ method there is the core code that creates the ontology:

- for each Class node, a new ontology class is created
- for each assignment found in the body of the class (the properties of that class) a new child class is created:
    - all properties are DataProperty except ‘body’ and ‘parameters’; these two are ObjectProperties

This is the code that loops through all the statements and filters out all except the assignments:

```
# for all statements that are assignments
   for a in (stmt for stmt in node.body if type(stmt) == ast.Assign):
       # read the tuple values and create the according property
       for val in a.value.elts:
           self.create_property(val)
```

The method ‘create_property()’ takes care to rename the property ‘name’ to ‘jname’ to avoid conflicts with the predefined ‘name’ attribute of ontology instances.

Ontology pictures

This section shows the pictures of Protégé with the loaded ontology.

We can see:

- The class hierarchy in Figure 1
- The data property hierarchy in Figure 2
- The object property hierarchy in Figure 3

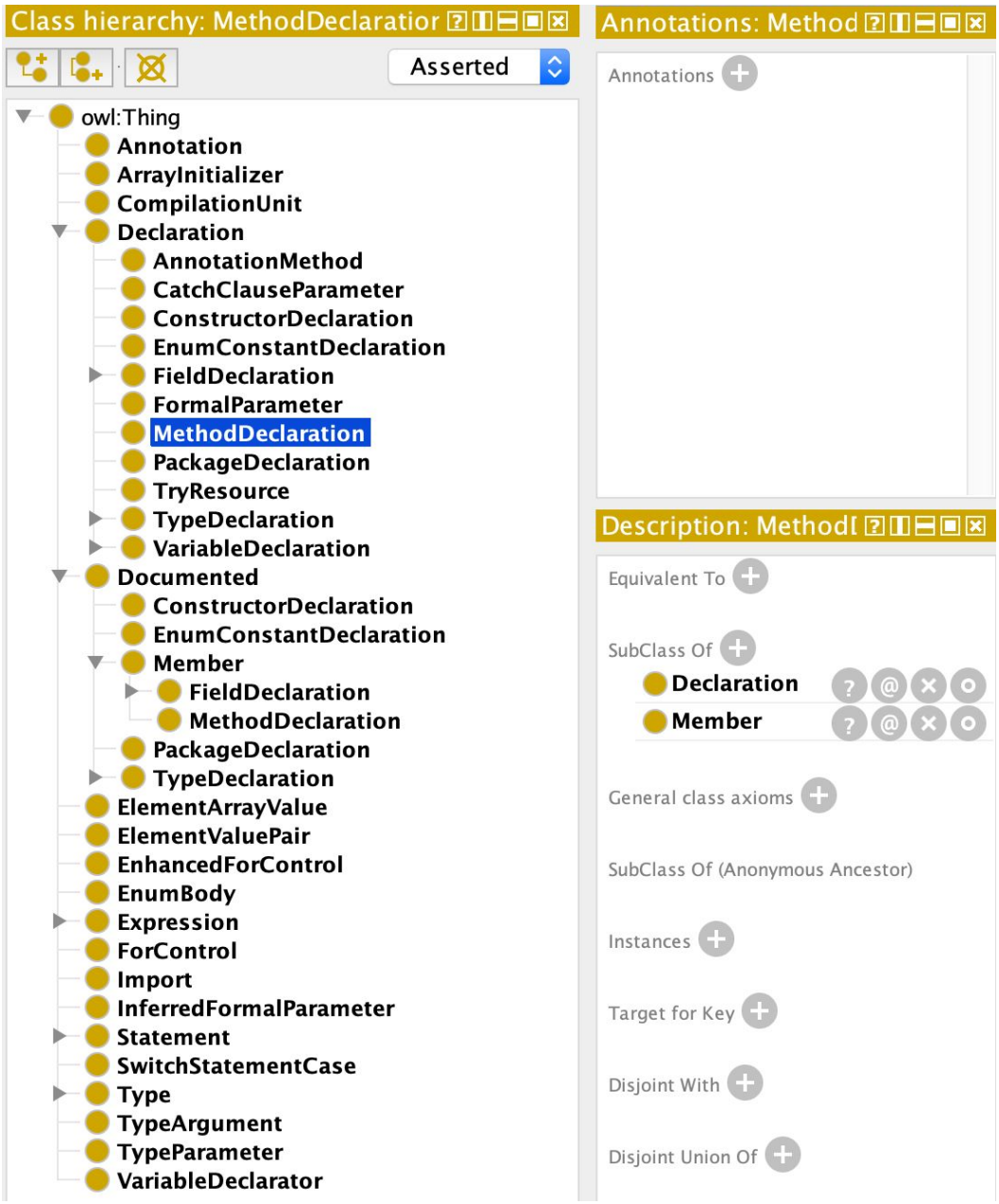

Figure 1. The class hierarchy of the ontology

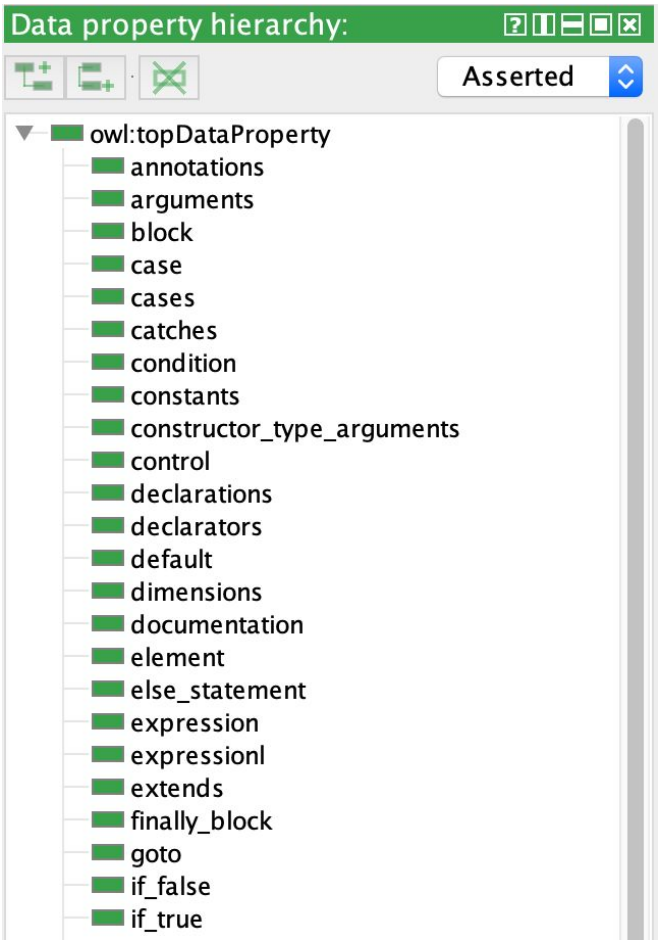

Figure 2. partial data property hierarchy

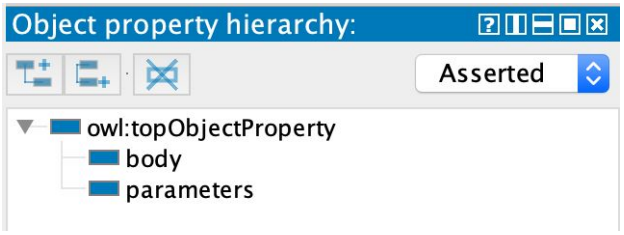

Figure 3. Object property hierarchy

As we expected, the Class hierarchy shows a nested structure.

I will take one class, as a sample, and check that what is inside ‘tree.py’ is correctly represented: the following code is the python code found inside ‘tree.py’ for “MethodDeclaration”:

```
class MethodDeclaration(Member, Declaration):
    attrs = ("type_parameters", "return_type", "name", "parameters" \
                               , "throws", "body")
```

We notice that MethodDeclaration is found in two places:

- Under the node Declaration (highlighted in Figure 1)
- Under the node Member

We now know that “MethodDeclaration” is correctly represented.

Two last checks are:

- we should only have two object property; as we can see in Figure 3, it checks out.
- all the other properties are DataProperty; as we can see in Figure 2, it checks out (the figure shows a partial list because we have many more properties).

Ontology metrics

Find in Fig. 4 the metrics for the created ontology.

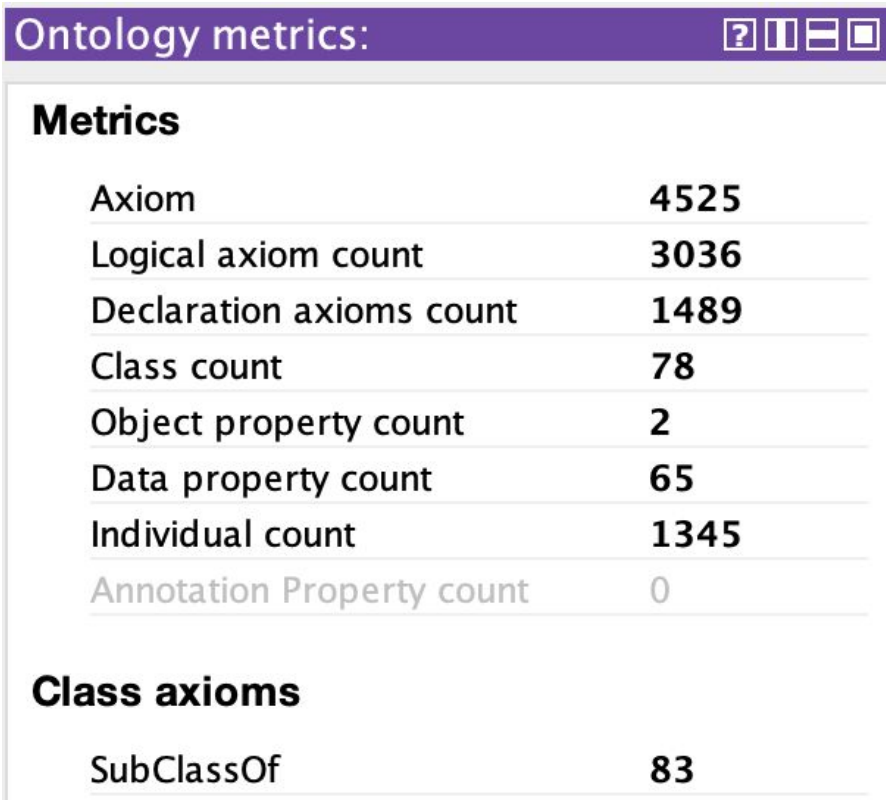

Figure 4. Ontology metrics

Populate ontology

In the previous section, we created the ontology classes and properties.

In this section, we are going to populate the ontology with instances (classes, class members, statements and method parameters)

Code for populating the ontology

The python source code produced for this section is ‘individ-creator.py’.

Firstly I will explain the high level of how the program works:

- it loads the ontology from ‘tree.owl’
- it walks the android-chess-master folder parsing all the java files
- for each parsed AST it processes all the types inside it and fills the ontology
- it saves the ontology to ‘tree2.owl’

What follows is the skeleton of the functions which I will further explain later:

```
def main():

    # code to load the recipient ontology from tree.owl
    [...]

    def process_callable_declaration(callable, callable_instance):
        [...]

    def process_class(class_node, package_name):
        [...]

    def process_types(types, package_name):
        [...]

    # loops through all the java sources and call the three functions above
    [...]
    # save the filled ontology
    onto.save('tree2.owl')
```

The previous excerpt of code shows the main building blocks of the program:

- an initial phase for loading the ontology
- three nested functions that process the AST of one parsed java file
- the final block where we can find:
    - the loop on all the java sources and related java parsing
    - for every parsed AST the invocation

The part that is actually producing the instances is contained in the three nested functions. The following is a break down of what those functions do:

The function `process_type(...)` is a utility function that recursively calls itself in order to process all classes contained in one AST. When it finds a 'ClassDeclaration' it calls `process_class(...)`

The function `process_class(...)` loops through the body of the class looking for the following types:

- MethodDeclaration
- ConstructorDeclaration
- FieldDeclaration

When it finds a FieldDeclaration it will directly create the ontology instance needed. For the other two declarations, it will delegate the creation of the instances to `process_callable_declaration(...)`

The function `process_callable_declaration(...)` exists only to be reused by the previous function: it creates the needed ontology instances and it works for both for MethodDeclaration and ConstructorDeclaration.

I should mention that I wrote a small helper class `PrettyPrint` to better show the class tree. This helped me during the development because the information is shown on the screen in a nice indented and very readable way.

This is an example of the output:

```
parsing ./chess/PGNEntry.java
   processing jwtc.chess.PGNEntry
      field jwtc.chess.PGNEntry._sMove[field]
      field jwtc.chess.PGNEntry._sAnnotation[field]
      field jwtc.chess.PGNEntry._move[field]
      constructor jwtc.chess.PGNEntry.$constructor$.PGNEntry(String,String,int)
parsing ./chess/PGNColumns.java
   processing jwtc.chess.PGNColumns
      field jwtc.chess.PGNColumns.CONTENT_TYPE[field]
      field jwtc.chess.PGNColumns.CONTENT_ITEM_TYPE[field]
      field jwtc.chess.PGNColumns.DEFAULT_SORT_ORDER[field]
      field jwtc.chess.PGNColumns.WHITE[field]
```

Statistics for the individuals

After populating the individuals we have these instances:

- ClassDeclaration, 11 instances (Figure 5)
- MethodDeclaration, 152 instances (Figure 6)
- FieldDeclaration, 105 instances (Figure 7)

During the creation of the instances, I decided to put a meaningful value also in the predefined ‘name’ property of the instance. As you can see in the following figures, the visual feedback is much more helpful than an auto-assigned identifier.

Because the property ‘name’ must be unique, I needed some extra careful computation with the MethodDeclaration: in the case of java override, we can have multiple methods with the same name but different arguments. Notice in figure 7 the MethodDeclarations explicitly shows the arguments between parenteses.

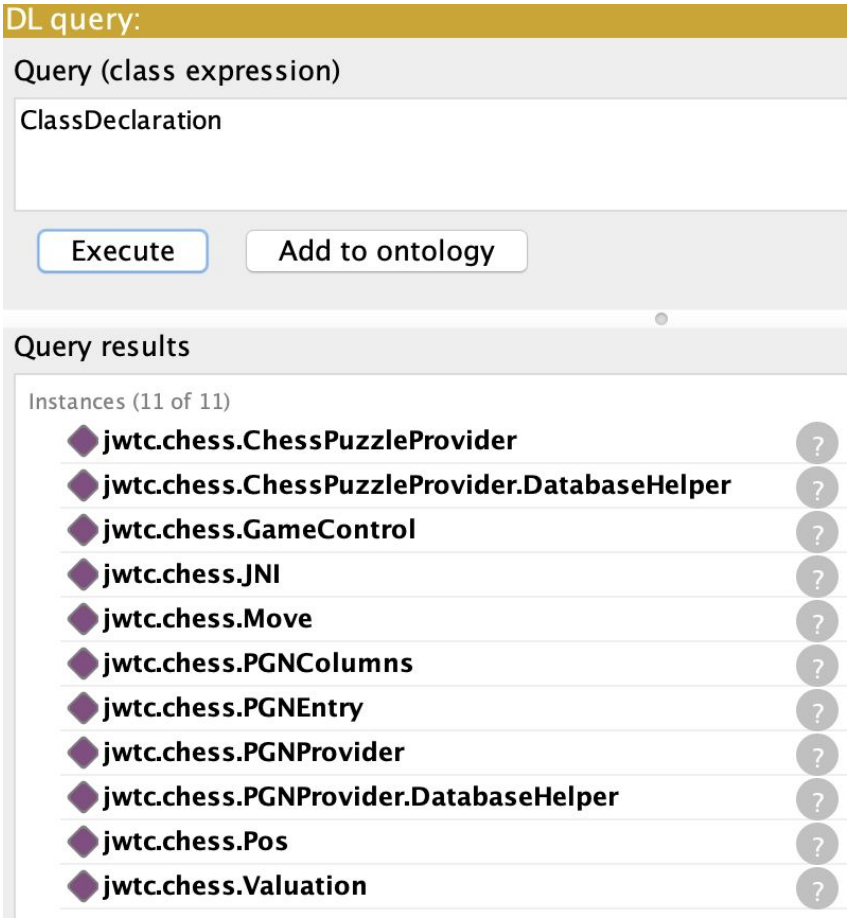

Figure 5. ClassDeclaration instances

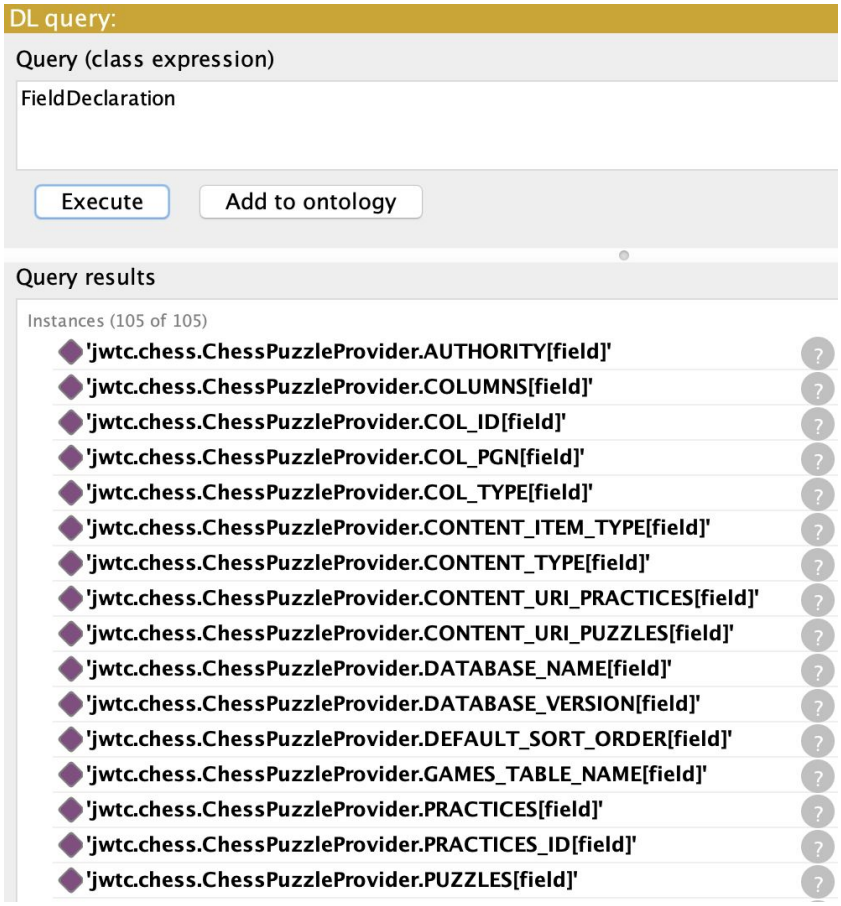

Figure 5. FieldDeclaration instances (partial list)

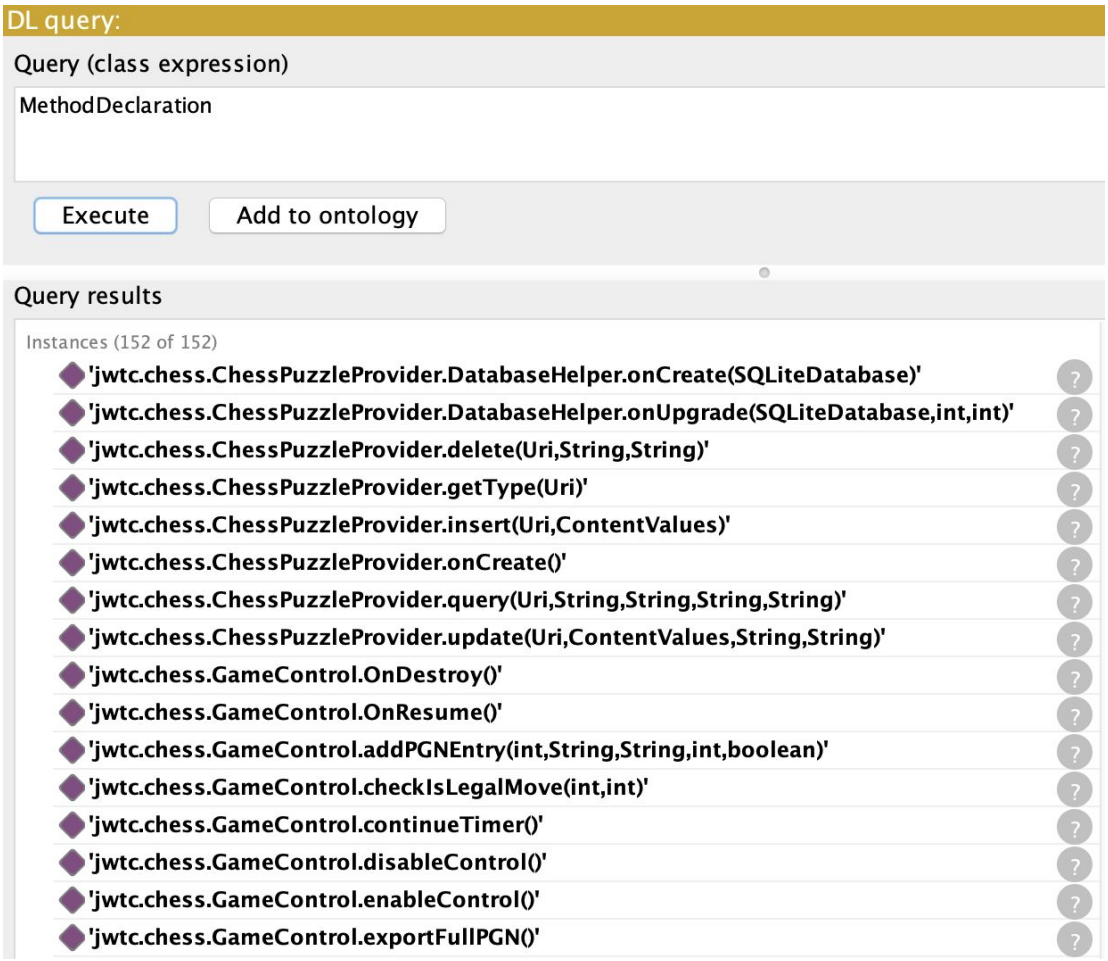

Figure 6. MethodDeclaration instances (partial list)

Find bad smells

In this section, I will use ontologies to find bad smells in code.

The main task is to encode Sparql queries that detects the target bad smells.

Code to detect bad smells

The python code to implement the requested functionalities is quite simple.

The most difficult task of this section was to implement the SPARQL query: in fact, if it worked in Protégé it didn't mean that the exact same query would work in python rdflib.

At a high level the program works with the following basic steps:

- It loads the ontology

- it opens the file ‘log.txt’ in write mode
- it defines a dictionary where the keys are the query names and the values are the SPARQL queries
- loops on the previous dictionary running the queries and appending the output to the console and also to the file ‘log.txt’
- it handles a special case for ‘DataClass’ bad smell; I did not found a way to do it with just one query, so I did it with two queries and some python code

Be aware that the queries have usually three fields:

- cn, class name
- mn, method name
- tot, total occurrences count

In some cases, the query has less than the previously mentioned field list. This is why I will employ the use of the helper function `get(...)` in order to fall back to an empty string for the missing fileds. With this technique, I can reuse more code.

The most notable functions are:

- `rdfquery(query)` is a function that accepts a SPARQL query string, run the query in the loaded graph and return the result
- `get(dictionary,fieldname)` is a simple function that fallback to an empty string if the field is not contained in the dictionary, as briefly explained above
- `query_to_set(query)` this is a one-liner function used only by the DataClass bad smell; it converts the SPARQL resultset to a python set
- `queries()` it returns a dictionary with the name-query pair. I initially had the queries into separate files but I preferred to change to a solution that did not rely on external files for reading the queries

The first lines of the main() function deal with creating the file ‘log.txt’.

The only other notable piece of code is about handling the special case of ‘DataClass’ smell; it is composed of two queries:

- the first, called yes_setget, returns the distinct list of all the class that contains at least one getter or setter
- the second, called no_setget, returns the distinct list of all the classes that contain at least one method that is not either a getter or a setter

Computing the subtraction of the two queries above yield the list of classes with only setters and getters (i.e. the requested bad smell); this is the loop that prints these classes:

```
for cn in query_to_set(yes_setget) - query_to_set(no_setget):
       log(f"  {cn} :: ()")
```

The following is the loop that handles all other SPARQL queries:

```
for name, query in queries().items():
       log(name + ':')
       for r in rdfquery(query):
           log(f"  {get(r, 'cn')} :: {get(r, 'mn')} ({get(r, 'tot')})")
       log()
```

Bad smells statistics

Int this section I report the number of occurrences of each bad smell found in the code, as well as the list of code entities containing each smell for the considered project directory.

We are requested to look for eight bad smells; table 1 shows the number of occurrences of such bad smells.

Bad smell	**number of occurrences**
DataClass	1
LongMethod	10
LongConstructor	0
LargeClass	3
MethodWithSwitch	8
ConstructorWithSwitch	0
MethodWithLongParameterList	6
ConstructorWithLongParameterList	0

Table 1. number of occurrences of each bad smell

The following list shows the bad smell name and the related entities that are affected by that particular smell.

```
DataClass:
  Valuation :: ()

LongMethod:
  GameControl :: loadPGNMoves (97)
  JNI :: initFEN (88)
  GameControl :: requestMove (87)
  JNI :: initRandomFisher (87)
  JNI :: newGame (35)
  PGNProvider :: insert (31)
  GameControl :: loadPGNHead (26)
  GameControl :: getDate (26)
  ChessPuzzleProvider :: query (25)
  ChessPuzzleProvider :: insert (20)

LongConstructor:
  No occurrences detected

LargeClass:
  GameControl ::  (63)
  JNI ::  (44)
  Move ::  (21)

MethodWithSwitch:
  PGNProvider :: query (1)
  PGNProvider :: delete (1)
  ChessPuzzleProvider :: update (1)
  ChessPuzzleProvider :: getType (1)
  PGNProvider :: getType (1)
  ChessPuzzleProvider :: delete (1)
  ChessPuzzleProvider :: query (1)
  PGNProvider :: update (1)

ConstructorWithSwitch:
  No occurrences detected

MethodWithLongParameterList:
  JNI :: setCastlingsEPAnd50 (6)
  DatabaseHelper :: onUpgrade (6)
  PGNProvider :: query (5)
  GameControl :: addPGNEntry (5)
  GameControl :: requestMove (5)
  ChessPuzzleProvider :: query (5)

ConstructorWithLongParameterList:
  No occurrences detected
```

Appendix: Python code

```
------------------
```

onto-creator.py

```
import ast

from owlready2 import *


def main():
    with open("tree.py", "r") as source:
        tree = ast.parse(source.read())

    onto = get_ontology("http://usi.ch/giacomelli/Knowledge_Analysis_and_Management.owl")

    analyzer = Analyzer(onto)
    analyzer.visit(tree)

    onto.save('tree.owl')


class Analyzer(ast.NodeVisitor):
    def __init__(self, onto):
        self.onto = onto

    def visit_ClassDef(self, node: ast.ClassDef):
        x: ast.Name
        print(f'visit_ClassDef {node.name} {[x.id for x in node.bases]}')

        bases = tuple([self.decodeBase(b) for b in node.bases])
        with self.onto:
            cl = types.new_class(node.name, bases)

        # for all statements that are assignments
        for a in (stmt for stmt in node.body if type(stmt) == ast.Assign):
            # read the tuple values and create the according property
            for val in a.value.elts:
                self.create_property(val)

    def create_property(self, val):
        id = val.s
        if id == 'name':
            id = 'jname'
        bases = (ObjectProperty,) if id == 'body' or id == 'parameters' else (DataProperty,)
        print(f'  Creating {id} {bases}')
        with self.onto:
            types.new_class(id, bases)

    def decodeBase(self, base):
        return Thing if base.id == 'Node' else self.onto[base.id]


if __name__ == "__main__":
    main()
```

---

individ-creator.py

```
import javalang
from owlready2 import *


def params_to_str(method_declaration):
    res = ','.join([p.type.name for p in method_declaration.parameters])
    return res
```

```
class PrettyPrint:
    def __init__(self):
        self.level = 0

    def print(self, *arg):
        print(' ' * (self.level * 3), end='')
        print(*arg)

    def inc(self):
        self.level += 1

    def dec(self):
        self.level -= 1


def main():
    onto = get_ontology("file://tree.owl")
    onto.load()

    source_folder_path = "./chess"
    if len(sys.argv) > 1:
        source_folder_path = sys.argv[1]

    if not os.path.isdir(source_folder_path):
        print('Folder ', source_folder_path, 'not found')
        exit(0)

    source_files_path = [f for f in [os.path.join(source_folder_path, ent) for ent in
os.listdir(source_folder_path)]
                         if os.path.isfile(f)]
    pp = PrettyPrint()

    def process_callable_declaration(callable, callable_instance):
        pp.inc()

        for dec in callable.parameters:
            type_name = type(dec).__name__
            dec_instance = onto[type_name]()
            callable_instance.parameters.append(dec_instance)

        if callable.body is None:
            pp.print('None!')
        else:
            for path, st in callable:
                if isinstance(st, javalang.parser.tree.Statement):
                    type_name = type(st).__name__
                    dec_instance = onto[type_name]()
                    callable_instance.body.append(dec_instance)

        pp.dec()

    def process_class(class_node, package_name):
        pp.inc()
        class_instance = onto["ClassDeclaration"]()
        class_fqn = package_name + '.' + class_node.name
        pp.print(f'processing {class_fqn}')
        class_instance.name = class_fqn
        class_instance.jname.append(class_node.name)
        pp.inc()
        for dec in class_node.body:
            type_name = type(dec).__name__
            if type_name == 'MethodDeclaration':
                dec_instance = onto[type_name]()
                class_instance.body.append(dec_instance)
                method_fqn = class_fqn + '.' + dec.name + f'({params_to_str(dec)})'
                pp.print(f'method {method_fqn}')
                dec_instance.jname.append(dec.name)
                dec_instance.name = method_fqn
                process_callable_declaration(dec, dec_instance)
```

```
            elif type_name == 'FieldDeclaration':
                for f in dec.declarators:
                    field_fqn = class_fqn + '.' + f.name + '[field]'
                    pp.print(f'field {field_fqn}')
                    dec_instance = onto[type_name]()
                    class_instance.body.append(dec_instance)
                    dec_instance.jname.append(f.name)
                    dec_instance.name = field_fqn
            elif type_name == 'ConstructorDeclaration':
                constructor_fqn = class_fqn + '.$constructor$.' + dec.name +
f'({params_to_str(dec)})'
                pp.print(f'constructor {constructor_fqn}')
                dec_instance = onto[type_name]()
                class_instance.body.append(dec_instance)
                dec_instance.jname.append(dec.name)
                dec_instance.name = constructor_fqn
                process_callable_declaration(dec, dec_instance)
            else:
                pass
                # pp.print(f'else {type_name}')
                # process_types(dec, class_fqn)

        pp.dec()
        pp.dec()

    def process_types(types, package_name):
        for node in types:
            if type(node) is javalang.tree.ClassDeclaration:
                process_class(node, package_name)
                process_types(node.body, package_name + '.' + node.name)

    for source_path in source_files_path:
        with open(source_path, "r") as file_handle:
            source = file_handle.read()
        pp.print('parsing', source_path)
        tree = javalang.parse.parse(source)

        process_types(tree.types, tree.package.name)

    onto.save('tree2.owl')


if __name__ == '__main__':
    main()
```

---

bad-smells.py

```
import rdflib
import rdflib.plugins.sparql as sq

g = rdflib.Graph()
g.load("tree2.owl")


def rdfquery(query):
    namespace = "http://usi.ch/giacomelli/Knowledge_Analysis_and_Management.owl#"
    q = sq.prepareQuery(query, initNs={"tree": namespace})
    return g.query(q)


def get(dictionary, fieldname):
    if fieldname in dictionary.labels:
        return dictionary[fieldname]
    return ''
```

```
def query_to_set(query):
    return {r.cn for r in rdfquery(query)}


def main():
    with open('log.txt', 'w') as f:
        def log(*args):
            print(*args)
            f.write(' '.join(args) + '\n')

        # DataClass bad smell is a special case
        log('DataClass:')
        yes_setget, no_setget = data_class_queries()
        for cn in query_to_set(yes_setget) - query_to_set(no_setget):
            log(f"  {cn} :: ()")
        log()

        # the rest of the bad smells
        for name, query in queries().items():
            log(name + ':')
            for r in rdfquery(query):
                log(f"  {get(r, 'cn')} :: {get(r, 'mn')} ({get(r, 'tot')})")
            log()


def queries():
    return {
        'LongMethod': """
    PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
    PREFIX owl: <http://www.w3.org/2002/07/owl#>
    PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
    PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
    PREFIX tree: <http://usi.ch/giacomelli/Knowledge_Analysis_and_Management.owl#>
    SELECT ?cn ?mn (COUNT(*) AS ?tot) WHERE {
                ?c a tree:ClassDeclaration .
                ?c tree:jname ?cn .
                ?c tree:body ?m .
                ?m a tree:MethodDeclaration .
                ?m tree:jname ?mn .
                ?m tree:body ?st .
                ?st a/rdfs:subClassOf* tree:Statement .
            }

    GROUP BY ?cn ?mn
    HAVING (COUNT(*) >= 20)
    ORDER BY DESC(COUNT(*))
    """
        , 'LongConstructor': """
    PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
    PREFIX owl: <http://www.w3.org/2002/07/owl#>
    PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
    PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
    PREFIX tree: <http://usi.ch/giacomelli/Knowledge_Analysis_and_Management.owl#>
    SELECT ?cn ?mn (COUNT(?st) AS ?tot) WHERE {
                ?c a tree:ClassDeclaration .
                ?c tree:jname ?cn .
                ?c tree:body ?m .
                ?m a tree:ConstructorDeclaration .
                ?m tree:jname ?mn .
                ?m tree:body ?st .
                ?st a/rdfs:subClassOf* tree:Statement .
            }

    GROUP BY ?cn ?mn
    HAVING (COUNT(?st) >= 20)
    ORDER BY DESC(COUNT(*))
    """
        , 'LargeClass': """
```

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX tree: <http://usi.ch/giacomelli/Knowledge_Analysis_and_Management.owl#>
SELECT ?cn (COUNT(*) AS ?tot) WHERE {
            ?c a tree:ClassDeclaration .
            ?c tree:jname ?cn .
            ?c tree:body ?m .
            ?m a tree:MethodDeclaration .
        }

GROUP BY ?cn
HAVING (COUNT(*)  >= 10)
ORDER BY DESC(COUNT(*))
"""
    , 'MethodWithSwitch': """
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX tree: <http://usi.ch/giacomelli/Knowledge_Analysis_and_Management.owl#>
SELECT ?cn ?mn (COUNT(*) AS ?tot) WHERE {
            ?c a tree:ClassDeclaration .
            ?c tree:jname ?cn .
            ?c tree:body ?m .
            ?m a tree:MethodDeclaration .
            ?m tree:jname ?mn .
            ?m tree:body ?st .
            ?st a tree:SwitchStatement .
        }

GROUP BY ?cn ?mn
HAVING (COUNT(*) >= 1)
ORDER BY DESC(COUNT(*))
"""
    , 'ConstructorWithSwitch': """
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX tree: <http://usi.ch/giacomelli/Knowledge_Analysis_and_Management.owl#>
SELECT ?cn ?mn (COUNT(*) AS ?tot) WHERE {
            ?c a tree:ClassDeclaration .
            ?c tree:jname ?cn .
            ?c tree:body ?m .
            ?m a tree:ConstructorDeclaration .
            ?m tree:jname ?mn .
            ?m tree:body ?st .
            ?st a tree:SwitchStatement .
        }

GROUP BY ?cn ?mn
HAVING (COUNT(*) >= 1)
ORDER BY DESC(COUNT(*))
"""
    , 'MethodWithLongParameterList': """
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX tree: <http://usi.ch/giacomelli/Knowledge_Analysis_and_Management.owl#>
SELECT ?cn ?mn (COUNT(*) AS ?tot) WHERE {
            ?c a tree:ClassDeclaration .
            ?c tree:jname ?cn .
            ?c tree:body ?m .
            ?m a tree:MethodDeclaration .
            ?m tree:jname ?mn .
            ?m tree:parameters ?pa .
```

```
            }

        GROUP BY ?cn ?mn
        HAVING (COUNT(*) >= 5)
        ORDER BY DESC(COUNT(*))
        """
            , 'ConstructorWithLongParameterList': """
        PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
        PREFIX owl: <http://www.w3.org/2002/07/owl#>
        PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
        PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
        PREFIX tree: <http://usi.ch/giacomelli/Knowledge_Analysis_and_Management.owl#>
        SELECT ?cn ?mn (COUNT(*) AS ?tot) WHERE {
                    ?c a tree:ClassDeclaration .
                    ?c tree:jname ?cn .
                    ?c tree:body ?m .
                    ?m a tree:ConstructorDeclaration .
                    ?m tree:jname ?mn .
                    ?m tree:parameters ?pa .
            }

        GROUP BY ?cn ?mn
        HAVING (COUNT(*) >= 5)
        ORDER BY DESC(COUNT(*))
        """

        }


    def data_class_queries():
        return ["""
        PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
        PREFIX owl: <http://www.w3.org/2002/07/owl#>
        PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
        PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
        PREFIX tree: <http://usi.ch/giacomelli/Knowledge_Analysis_and_Management.owl#>
        SELECT distinct ?cn WHERE {
                    ?c a tree:ClassDeclaration .
                    ?c tree:jname ?cn .
                    ?c tree:body ?m .
                    ?m a tree:MethodDeclaration .
                    ?m tree:jname ?mn .
                    FILTER regex(?mn, "^get.*|^set.*")
            }
        """, """
            PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
            PREFIX owl: <http://www.w3.org/2002/07/owl#>
            PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
            PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
            PREFIX tree: <http://usi.ch/giacomelli/Knowledge_Analysis_and_Management.owl#>
            SELECT distinct ?cn WHERE {
                        ?c a tree:ClassDeclaration .
                        ?c tree:jname ?cn .
                        ?c tree:body ?m .
                        ?m a tree:MethodDeclaration .
                        ?m tree:jname ?mn .
                        FILTER ( !EXISTS {
                            FILTER regex(?mn, "^get.*|^set.*")
                        } )
                    }
        """]


    if __name__ == '__main__':
        main()
```