Knowledge Management and Analysis
Project 01: Bad smells

Davide Mammarella

Section 1 - Ontology Creation

The first step is to create an ontology for every Java entity (Class, Method, Field, Statement, etc.) in the `tree.py` file present in the **javalang** python library. The file that creates the ontology is `onto-creator.py` and the packages **Owlready2** and **AST** are needed. In practice, `onto-creator.py` implements a custom subclass of the visit class from the AST package to parse the ast of the `tree.py` file and generate an ontology.

Structure of the created ontology

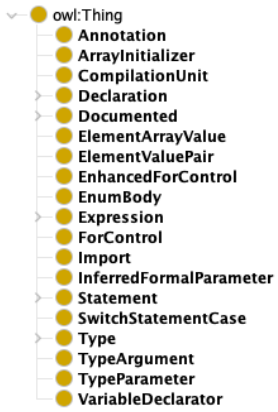

Figure 1: Class Hierarchy of the created ontology.

The resulting ontology, or `tree.owl`, reflects the **Java language specification** and contains its related entities in a nested manner. During the creation of the ontology, a class was created for each ClassDef object in the ast of `tree.py` and the resulting hierarchy can be seen in Figure 1. For completeness, the nested entities are shown in Figure 2.

Taking into consideration the simplest entities, for an intuitive comparison, we can see for example that the class "Statement" contains sublcasses referring to Java loop instructions: "DoStatement", "ForStatement", "IfStatement", "SwitchStatement" and "WhileStatement".

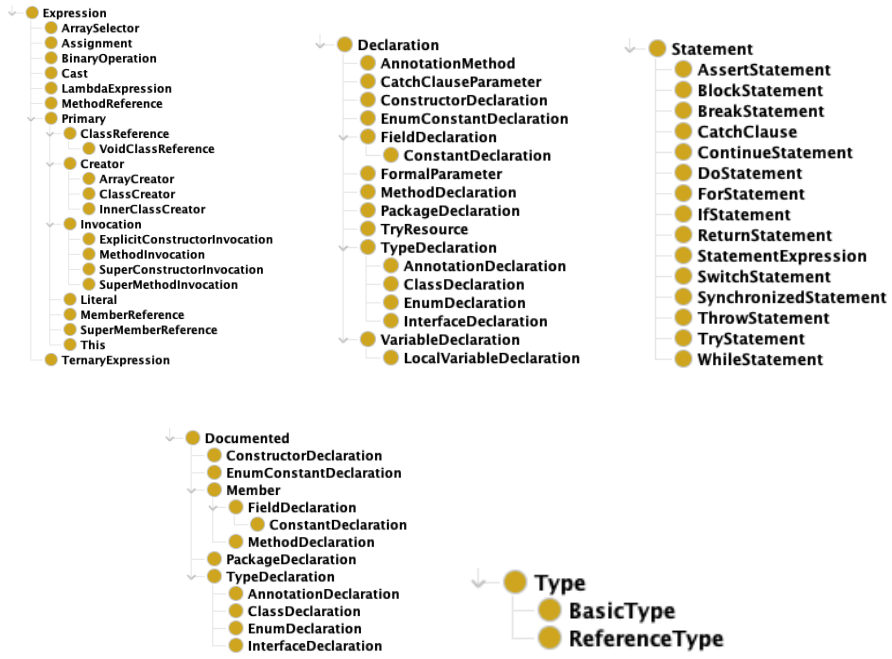

Figure 2: Nested Entities of the Class Hierarchy

Furthermore, during the creation of the ontology, the relevant properties for the ontology are extracted when visiting each ClassDef object and these are all considered as Data Properties with the exception of "body" and "parameters" which are considered as Object Properties. This is implemented in the file `onto-creator.py`, and reflects the results obtained, which can be seen in Figure 3, since the Data Properties (a),(b) are multiple and the Object Properties (c) are only "body" and "parameters".

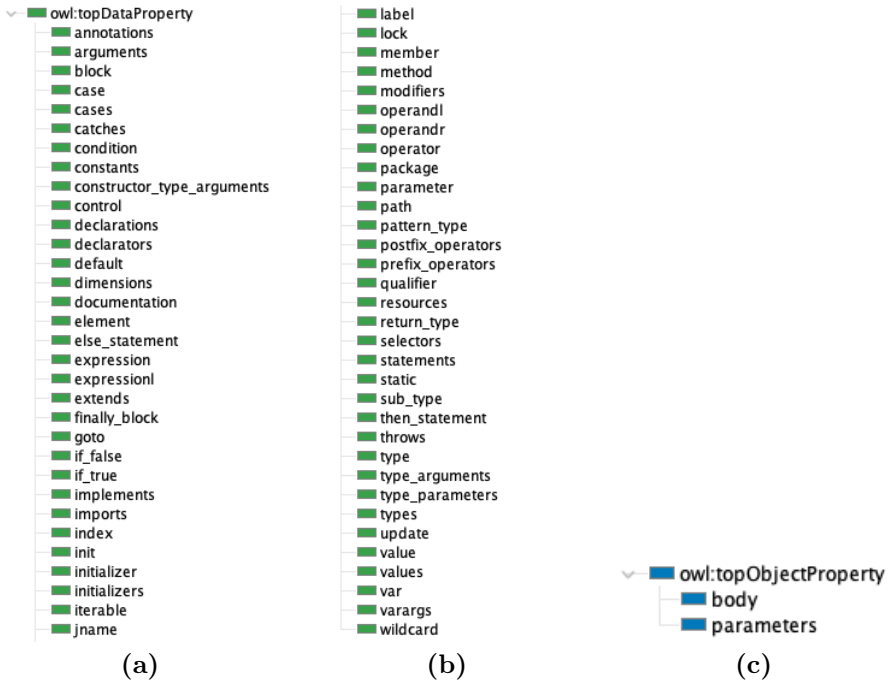

Figure 3: Data Properties (a),(b) and Object Properties (c)

Number of created classes and properties

The total number of created classes and properties are shown in Table 1. These metrics have been obtained, and can be verified, by opening the `tree.owl` file with the Protegé application and selecting: *Window* > *Views* > *Ontology views* > *Ontology metrics*.

Type	Number
Class count	78
Object property count	2
Data property count	65

Table 1: Count of created classes and properties.

Section 2: Creation of ontology instances

Number of created instances

The ontology created in Section 1 is populated using the `individ-creator.py` file, which is responsible for generating individuals following the analysis of various Java files passed as input from an android application called **AndroidChess**. Each instance of **class, class member, statement** and **parameter** contained in the classes within the application is added into the `tree2.owl` ontology and the metrics relating to the number of individuals created are available in Table 2. The results are reproducible through the Protegé application, by opening the related ontology, going to the *DL Query* section, and making a query for *Instances* using as body of the query the name of the class whose metrics are desired.

Class	Number of created individuals
ClassDeclaration:	11
MethodDeclaration	152
ConstructorDeclaration	6
FieldDeclaration	105
BlockStatement	143
BreakStatement	23
CatchClause	8
ContinueStatement	4
DoStatement	2
ForStatement	6
IfStatement	125
ReturnStatement	106
StatementExpression	446
SwitchStatement	8
SynchronizedStatement	1
ThrowStatement	15
TryStatement	8
WhileStatement	10
FormalParameter	165

Table 2: Number of created instances per class.

Section 3: Bad smell detection

The ontology created and populated in the previous section, i.e. `tree2.owl` is analysed for bad smells by using **SPARQL** queries, through the **rdflib** package. The source code is in the `bad-smells.py` file and the detected bad smells are reported in Table 3.

Bad Smell	Count
Long methods	10
Long constructors	0
Large classes	3
Methods with switch statements	8
Constructors with switch statements	0
Methods with long parameter list	4
Constructors with long parameter list	0
Data Classes	1

Table 3: Bad smells (Total).

Finally, for each bad smell found, the metrics according to which it is considered a bad smell and the tables with more details on the **methods involved** and the **number of occurrences** are reported, excluding bad smells that have no occurrences.

Methods with 20 or more statements are considered **long methods**, and are listed in Table 4

Class Name	Method Name	Number of Occurrences
GameControl	loadPGNMoves	97
JNI	initFEN	88
JNI	initRandomFisher	87
GameControl	requestMove	76
JNI	newGame	35
PGNProvider	insert	31
GameControl	loadPGNHead	26
GameControl	getDate	26
ChessPuzzleProvider	query	25
ChessPuzzleProvider	insert	20

Table 4: Long methods.

Classes with 10 or more methods are considered **large classes**, and are listed in Table 5

Class Name	Number of Occurrences
GameControl	63
JNI	44
Move	21

Table 5: Large classes.

Methods with 1 or more switch statement in method or constructor body are considered **methods with switch statements**, and are listed in Table 6

Class Name	Method Name	Number of Occurrences
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 6: Methods with switch statements.

Methods with 5 or more parameters are considered **methods with long parameter list**, and are listed in Table 7

Class Name	Method Name	Number of Occurrences
JNI	setCastlingsEPAnd50	6
PGNProvider	query	5
ChessPuzzleProvider	query	5
GameControl	addPGNEntry	5

Table 7: Methods with long parameter list.

Classes with only getters and setters are considered **data classes**, and are listed in Table 8. In order to reproduce the results in the table, it is necessary to make two queries and filter their results. In practice, these two queries are executed within the code of `bad-smells.py` file: the first, whose result is in the variable `getset_classes_query_result`, allows us to obtain the number of filtered methods (i.e. methods containing get or set) for each class, while the second, whose result is in the variable `all_classes_query_result`, allows us to obtain the number of unfiltered methods (i.e. all methods) for each class. These two results are then compared and only classes containing the same number of filtered and unfiltered methods are identified as data classes.

Class Name	Filtered Method	Unfiltered Method
Valuation	1	1

Table 8: Data classes.