INVocD: Identifier Name Vocabulary Dataset

Simon Butler, Michel Wermelinger, Yijun Yu and Helen Sharp
Centre for Research in Computing, Department of Computing
The Open University, Milton Keynes, United Kingdom

***Abstract*—INVocD is a database of the identifier name declarations and vocabulary found in 60 FLOSS Java projects where the source code structure is recorded and the identifier name vocabulary is made directly available, offering advantages for identifier name research over conventional source code models. The database has been used to support a range of research projects from identifier name analysis to concept location, and provides many opportunities to researchers. INVocD may be downloaded from http://oro.open.ac.uk/36992**

***Index Terms*—identifier names; source code model; source code mining**

I. Introduction

Typical source code models, e.g. the Dagstuhl middle model (DMM) [1] and FAMIX [2], are AST based and focus on code structure with identifier names often recorded as attributes of AST nodes. The researcher interested in identifier names and identifier name vocabulary has then to extract identifier names from the source code model. In this paper we present INVocD (Identifier Name Vocabulary Database), a database model populated with the identifier name declarations in 60 FLOSS Java projects (including ArgoUML, Derby, Eclipse, Hibernate, Tomcat and Vuze) where the source code entities, and their structural relationships, are recorded and the identifier names and their component words are made directly available to the database user. The component words and identifier name records effectively form an index to the source code. For example, the component word `array` found in the identifier name `ANEWARRAY` (FindBugs) is linked to all other identifier names in INVocD containing the word `array`.

The 60 projects recorded in the database contain 5,091,000 program entities, including 626,000 field name, 1,238,000 method and 1,319,000 local variable declarations. There are some 831,000 unique identifier names, including imported types, constructed from 25,000 unique component words.

INVocD has been used to support a number of research papers including work on identifier name tokenisation [3] and the analysis of class name structure [4]. The infrastructure that created INVocD has also been used to support the development and evaluation of concept location techniques [5].

II. Schema

The INVocD schema has two principal elements. The first is the `PROGRAM_ENTITIES` table that is used to store entities found in the AST. The second is the small group of tables that record the identifier name vocabulary. The two are linked, allowing the identifier name vocabulary used in program entities to be recovered, and the program entities in which particular words or groups of words are used to be identified.

The model of source code used in `PROGRAM_ENTITIES` (see Figure 1) is predicated on the idea that a program consists of a set of program entities that may contain other program entities. Identifier names are declared as labels for program entities, and the majority of program entities are named. By recording program entities, their relationships and their associated identifier names, the model records the source code structure and location and relative relationships of the entities containing vocabulary found in the program.

Rows in the `PROGRAM_ENTITIES` table represent each identifier name declaration in the AST and any unnamed nodes, such as initialiser blocks and anonymous classes, that might contain further identifier name declarations.

The `PROGRAM_ENTITIES` table contains the columns `CONTAINER_UID` and `ENTITY_UID`, which record the hierarchical relationships between program entities. The UIDs are generated during parsing using an SHA1 digest of a unique string for each file created by concatenating the project name and version, the package name and the file name. A serial number is then appended to the hash for each program entity encountered in the AST for that file. It is possible to use database keys instead of UIDs to record relationships, but that requires each container entity to be stored in the database before its children, which constrains the implementation of the system writing to the database.

For each constructor and method, a unique method signature is recorded in the `METHOD_SIGNATURES` table. The signature is a string composed of the type names of the formal arguments in declaration order. For example, the signature (`String`; `int`) is recorded for the `java.lang.String` method `indexOf(String str, int fromIndex)`. The signatures distinguish between multiple constructors, and methods of the same name, including overloaded methods.

The *species* of the program entity (class, method, etc.) and Java *modifiers* (`private`, `static`, etc.) are recorded in separate tables. Species are stored in the `SPECIES` table and referenced from the `PROGRAM_ENTITIES` table. Modifiers are found in the `MODIFIERS` table and are cross-referenced through the `MODIFIERS_XREF` table, because there may be more than one modifier for each program entity. For example, fields that are class constants might be declared with the `public`, `final` and `static` modifiers.

The start and end line and column numbers of the program entity are recorded in the `PROGRAM_ENTITIES` table, and a foreign key references a unique file name recorded in the `FILES` table. File path information is not recorded, but may be derived, in part, from the package name (see Figure 2).

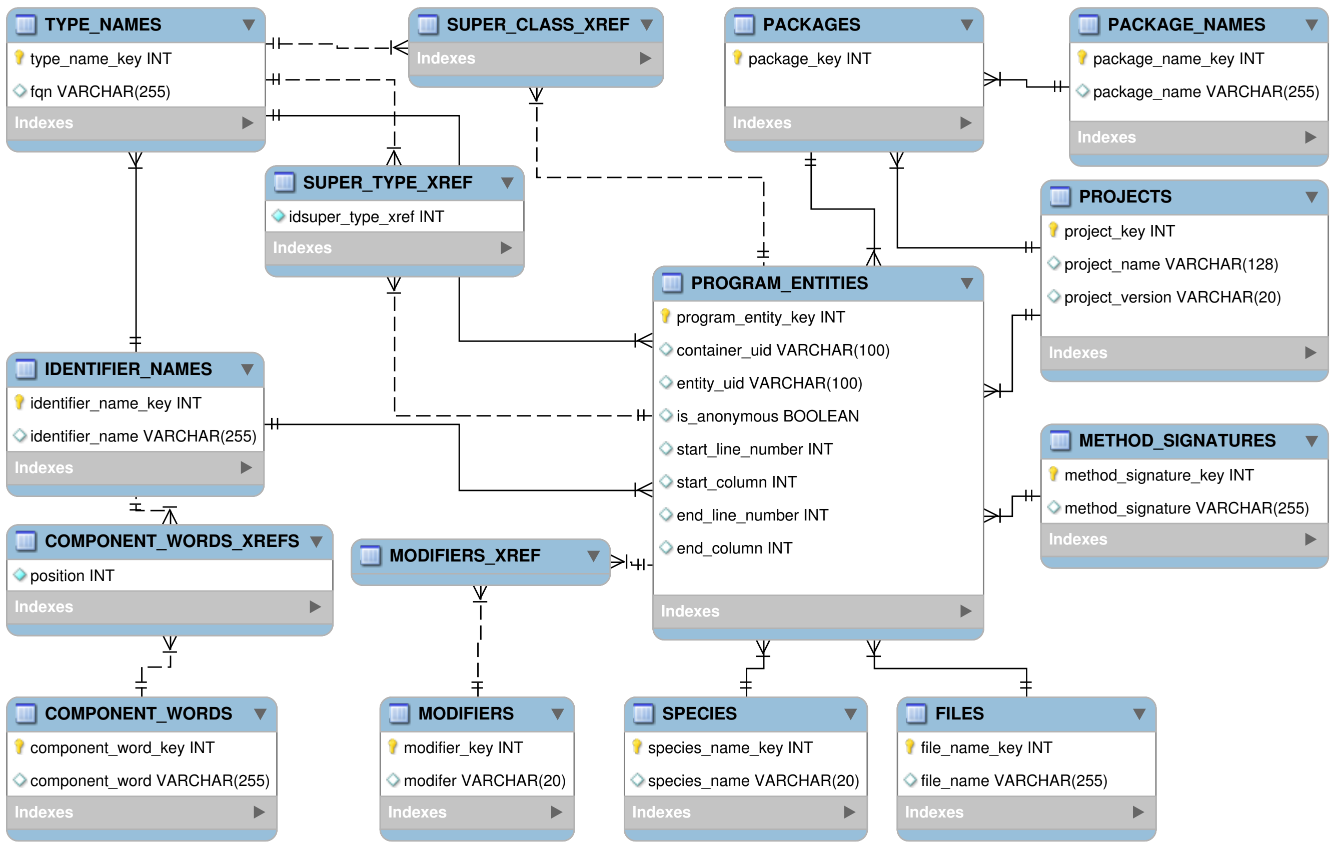

Fig. 1. The INVocD database model

Packages and projects are also containers like program entities. However, they share little common data with the programming language constructs that we record as program entities, e.g. they do not have types or modifiers. Details of packages and projects are stored in separate tables. Projects are recorded as a name and version pair, which allows the database to store multiple versions of the same project. Packages are recorded as unique packages belonging to a specific project in the PACKAGES table, and unique package names are recorded in the PACKAGE_NAMES table. This saves space when storing multiple versions of projects, where the package names remain largely unchanged between versions.

Unique identifier names are stored in the IDENTIFIER_NAMES table, which is referenced by both program entities and type names. The component words of identifier names are recorded as unique words in the COMPONENT_WORDS table and linked to identifier names through a cross reference table (COMPONENT_WORDS_XREFS) that records the position of the component word within the identifier name where the component word is found (indexing starts at 1). For example, keys for the identifier name ANEWARRAY and the component word array are stored in COMPONENT_WORDS_XREFS with the position 3. (An example SQL query using these tables is given in Figure 3.)

Type names are stored in the TYPE_NAMES table with a reference to the identifier name used to specify the type. The fully qualified name of the type may be recorded in the TYPE_NAME column. The resolution of fully qualified type names is experimental and is discussed in Section V. Generic types are not recorded. Any type with a generic type argument is recorded as the underlying type and the type argument ignored, e.g. ArrayList⟨String⟩ is recorded as ArrayList.

Type names involved in inheritance are cross referenced with program entities using two tables: SUPER_CLASS_XREF for class based inheritance (extends keyword), and SUPER_TYPE_XREF for type based inheritance (implements keyword, and extends when applied to interfaces).

III. Database Extraction

The tool that created the INVocD database, JIM or Java Identifier Miner, consists of 3 main components – a coordinating layer, a parser and a persistence layer – that in combination extract identifier names, component words and metadata concerning the location of the identifier name from source code and store it in an Apache Derby[1] database.

The coordination layer undertakes a recursive exploration of a directory tree supplied as a command line argument and parses every Java file found, except files named package-info.java, which contain only package level documentation comments, not source code.

[1]http://db.apache.org/derby/

The parser layer consists of two parsers: one for Java 1.5 and another for earlier versions. New language features were introduced in Java 1.5 including the new reserved word `enum` for enumerations. Consequently, source code which complies with older versions of Java and where `enum` is used as an identifier name will cause a Java 1.5 compliant parser to fail. The parsers use the Sun Java 1.5[2] grammar supplied with JavaCC[3], with some modifications to record identifier names in the resulting AST as well as metadata such as access modifiers. A parser for source files that conform to the Java 1.4 or earlier standard was created by removing keywords and productions related to features introduced in Java 1.5 from a second copy of the grammar. An attempt is made to parse a file using the Java 1.5 compliant parser. Should that fail, a further attempt is made using the alternative parser.

Some Java program entities do not have identifier names, e.g. initialiser blocks and anonymous classes. We record anonymous program entities with the identifier name '*#anonymous#*', which is not a legal Java identifier name. Similarly, program entities such as static initialisers that cannot have a type are given the type name '*#no type#*'. We adopted this method initially for debugging, but have found it a useful mechanism that allows analytical tools that extract data from INVocD to identify unnamed and untyped program entities.

The identifier names and metadata extracted from the AST node for each program entity are passed to the persistence layer, which manages the extraction of component words from identifier names using an updated version of identifier name tokenisation tool (INTT)[4] [3]. The program entity data, identifier names and component words are then stored in an Apache Derby database using the schema described earlier. The database is available for download from http://oro.open.ac.uk/36992 as a Derby database, a SQLite database and a SQL dump which can be imported into other RDBMSs.

IV. Opportunities

INVocD contains the identifier name declarations and source code structure of 60 Java projects. The projects were selected from a variety of domains including frameworks, IDEs, parser generators, project management applications, and servers[5], which allows the study of the vocabulary of groups of conceptually similar projects.

The identifier names in INVocD are drawn from a single programming language, and thus permit the investigation of the naming of single programming language entities across projects with the confidence that programming language is a consistent factor. For example, when comparing class identifier names the investigator can be confident that they are dealing with the linguistic influences of single inheritance only, and that field names, with the exception of parts of the reflection API, refer to entities only and not actions as well.

[2]All project versions in the database were released prior to Java 1.7.

[3]http://javacc.java.net/

[4]http://oro.open.ac.uk/28352/

[5]Full details can be found at http://www.facetus.org.uk/corpus.html

Corpora of both identifier names and component words may be extracted from INVocD according to specific criteria. It is possible, for example, to extract a single species of identifier name (see Figure 4 for an example SQL query) to study its structure – Java class names have been analysed using INVocD [4], and corpus of method names might be extracted to replicate studies by Høst and Østvold [6]. Public identifier names, and their component words, could be extracted to study the vocabulary used in APIs. The recorded structural relationships and type information in INVocD may also support the investigation of the anomalous word usage identified as lexicon bad smells by Abebe *et al.* [7].

The database contains a faithful record of the location, type, modifier and species for each identifier name declaration, as well as the component words identified by INTT. All the information can be used for the testing of new tokenisation and abbreviation expansion algorithms, e.g. using species or type information to support both processes, and the existing tokenisations as a baseline.

The storage of the component word vocabulary allows a single word to be used to identify all the identifier names in which it appears, and even allows the query to specify the position a word occurs in. For example, Figure 3 contains a query to recover all identifier names that begin with the component word 'array'.

The model of source code in INVocD also provides supports for developing vocabulary based search techniques. Dilshener and Wermelinger [5] used the infrastructure to create a database of identifier names for proprietary source code to look for vocabulary common to change requests and source code. They were then able to use the database to identify rapidly the source code locations of source terms. The example query in Figure 2 illustrates the retrieval of location information for a given identifier name.

The INVocD schema can be used to store representations of source code vocabulary from programming languages other than Java. The `MODIFIERS` and `SPECIES` tables can be extended to reflect other programming language constructs, and the tables for packages and package names used to record C++ or .NET namespaces. For programming languages without namespaces a dummy package might be recorded.

V. Limitations

There are three principal limitations to the information stored in INVocD. The first is that the database schema was developed for the study of identifier names, not data flow, and hence records only identifier declarations, not usage. The second is the recording of qualified type names, which is experimental. The third is the processing of identifier names to extract vocabulary.

Typical Java builds are automated with tools like Ant and Maven that are configured with often complex build paths indicating where external Java classes and libraries are found, and external symbols may be resolved. Our intention was to create a single pass tool that processed source code only. Hence, the resolution of external names is imprecise. External

```
select pn.package_name, f.file_name, pe.start_line_number from SVM.PROGRAM_ENTITIES pe
  join SVM.PACKAGES pkg on pkg.package_key=pe.package_key_fk
  join SVM.PACKAGE_NAMES pn on pn.package_name_key=pkg.package_name_key_fk
  join SVM.FILES f on f.file_name_key=pe.file_name_key_fk
  join SVM.IDENTIFIER_NAMES id on id.identifier_name_key=pe.identifier_name_key_fk
  join SVM.SPECIES sp on sp.species_name_key=pe.species_name_key_fk
  join SVM.PROJECTS pr on pr.project_key=pe.project_key_fk
  where pr.project_name='xom' and sp.species_name='method' and id.identifier_name='toString';
```

Fig. 2. Example SQL query to identify the start locations of `toString`() method declarations in XOM

```
select identifier_name from SVM.IDENTIFIER_NAMES id
  join SVM.COMPONENT_WORDS_XREFS cwx on id.identifier_name_key=cwx.identifier_name_key_fk
  join SVM.COMPONENT_WORDS cw on cw.component_word_key=cwx.component_word_key_fk
  where cw.component_word='array' and cwx.position=1;
```

Fig. 3. Example SQL query to recover unique identifier names beginning with the word 'array'

```
select identifier_name from SVM.IDENTIFIER_NAMES id
  join SVM.PROGRAM_ENTITIES pe on id.identifier_name_key=pe.identifier_name_key_fk
  join SVM.SPECIES sp on sp.species_name_key=pe.species_name_key_fk
  join SVM.PROJECTS pr on pr.project_key=pe.project_key_fk
  where pr.project_name='jedit' and sp.species_name='field';
```

Fig. 4. Example SQL query to recover field identifier name declarations in jEdit

symbols in Java may be indicated explicitly in the source code in import statements, or implied in 'starred' imports where all the public entities of a package are made available to the Java compiler to resolve. Alternatively, an external symbol may be found in the local package, or the default `java.lang` package. In the `TYPE_NAMES` table we record a reference to the unique identifier name used to specify the type in the source code file. Where the fully qualified type name is given in an import statement, we record it, otherwise we use heuristics – such as knowledge of public names in the local package, and fully qualified types already seen that may match starred imports – to try to identify the fully qualified type name. Consequently, the fully qualified type name may not be present, and may not be reliable even if it is present.

The extraction of component words from identifier names is not trivial and the accuracy of the identifier name tokeniser is a limiting factor on the quality of the component word vocabulary. INVocD was generated using our own tokeniser, INTT [3], which relies on dictionaries and can oversplit unrecognised abbreviations and words. Since INVocD includes the original identifier names, users can apply their own tokenisers.

VI. Conclusions

INVocD is a database of identifier name vocabulary, currently covering 60 FLOSS Java projects with over 830,000 unique identifiers naming over 5 million program entities. The component words form an index to the identifier names which are associated with the program entities where they were declared. The database schema allows access to identifier name vocabulary both independently and in conjunction with the source code structure, providing a low cost platform for imaginative identifier name vocabulary research, e.g. to improve naming conventions, program comprehension, feature location, abbreviation expansion, and tokenisation algorithms.

References

[1] T. Lethbridge, S. Tichelaar, and E. Plödereder, "The Dagstuhl middle metamodel: A schema for reverse engineering," *Electronic Notes in Theoretical Computer Science*, vol. 94, no. 0, pp. 7–18, 2004.

[2] S. Ducasse, N. Anquetil, U. Bhatti, A. C. Hora, J. Laval, and T. Girba, "MSE and FAMIX 3.0: an interexchange format and source code model family," INRIA, Tech. Rep., Nov 2011. [Online]. Available: http://hal.archives-ouvertes.fr/index.php?halsid=bvn39r6d5ucvb6e95asroiipj6&view_this_doc=hal-00646884&version=1

[3] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp, "Improving the tokenisation of identifier names," in *25th European Conf. on Object-Oriented Programming*, ser. Lecture Notes in Computer Science, M. Mezini, Ed., vol. 6813. Springer Berlin / Heidelberg, 2011, pp. 130–154.

[4] ——, "Mining Java class naming conventions," in *Proc. of the 27th IEEE Int'l Conf. on Software Maintenance*. IEEE, 2011, pp. 93–102.

[5] T. Dilshener and M. Wermelinger, "Relating developers concepts and artefact vocabulary in a financial software module," in *Proc. of the 27th IEEE Int'l Conf. on Software Maintenance*. IEEE, 2011, pp. 412–417.

[6] E. W. Høst and B. M. Østvold, "The Java programmer's phrase book." in *Software Language Engineering*, ser. LNCS, vol. 5452. Springer, 2008, pp. 322–341.

[7] S. L. Abebe, V. Arnaoudova, P. Tonella, G. Antoniol, and Y.-G. Gueheneuc, "Can lexicon bad smells improve fault prediction?" in *Proc. 19th Working Conference on Reverse Engineering (WCRE)*. IEEE, Oct. 2012, pp. 235 –244.