Mining Java Class Naming Conventions

Mining Java Class Naming Conventions

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*—Class names represent the concepts implemented in object-oriented source code and are key elements in program comprehension and, thus, software maintenance. Programming conventions often state that class names should be noun-phrases, but there is little further guidance for developers on the composition of class names. Other researchers have observed that the majority of Java class identifier names are composed of one or more nouns preceded, optionally, by one or more adjectives. However, no detailed analysis of class identifier name structure has been undertaken that could be leveraged to support program comprehension activities.**

We investigate the lexical and syntactic composition of Java class identifier names in two ways. Firstly, as others have done for C function and Java method names, we identify conventional patterns found in the use of parts of speech. Secondly, we identify the origin of words used in class names within the name of any super class and implemented interfaces to identify patterns of class name construction related to inheritance.

Through the analysis of 120,000 unique class names found in 60 open source projects we identify both common and project specific class naming conventions. We apply this knowledge in a case study of the mind-mapping tool Freemind to investigate whether class names that follow unconventional naming schemes are candidates for refactoring – either a name refactoring that conforms to established naming conventions within the code base, or refactoring of the class that results in conventionally named classes.

***Keywords*-Java class naming; identifier names;**

I. Introduction

Class identifier names represent the core entities and concepts encoded in object-oriented source code, and are vital to program comprehension [1], [2]. Programming conventions advise that developers choose 'meaningful' identifier names and that class identifier names should be nouns [3], [4] or descriptive nouns [4]. More advanced practitioner texts advocate a considered approach to identifier naming [5], and a variety of conventions of language use have arisen as a result of praxis [6].

Analysis of C function identifier names led to support for automated name refactoring [7]. Similar analysis of Java method identifier names found a link between identifier names and method implementation [8], which was successfully leveraged to identify candidates for name refactoring and to suggest possible refactorings [9].

Despite the importance of class identifier names for program comprehension, an essential activity in software development and maintenance, there is no detailed understanding of their structure. In this paper we present a survey of the class identifier names found in 16.5 MSLOC of open source Java projects. We analyse the identifier names to recover patterns of parts-of-speech (PoS) used in their construction and identify common grammatical structures. We also catalogue the repetition of the component words of super class and interface names in class identifier names. Such reuse is evident in the Java library, for example, where the `java.util` classes `HashSet` and `TreeSet` retain the name of the `Set` interface implemented by their common super class `AbstractSet`. However, little is known of the extent to which these types of pattern are replicated in production source code and under what circumstances.

Common, recognisable identifier naming conventions help support human program comprehension and provide a foothold for machine-based program comprehension techniques that extract knowledge from source code [10]. The class identifier naming patterns used in the wider Java community and within a given project are the mechanisms that developers are familiar with and use to communicate ideas in source code. This knowledge can be incorporated in IDE-based tools to help software developers create class identifier names that are more readily understood, and to provide quality assurance tools for software project managers that are an improvement on the functionality of current tools.

In a case study of Freemind, a Java mind-mapping tool, we investigate whether the class naming conventions identified can be applied to identify classes that are candidates for refactoring. Firstly where either the name might be refactored to conform to a more common naming convention, and, secondly, where the class might be refactored to form two or more conventionally named classes. We show that unusually structured class names can indicate the need to refactor a class identifier name and may also indicate the need for a class to be refactored.

The structure of the remainder of this paper is as follows: in Section II we explore related research and describe our methodology in Section III. In Section IV we give an account of our results and discuss our findings in Section V, before drawing our conclusions in Section VI.

II. Related Work

The key areas of research related to this paper concern the investigation and understanding of the structure of identifier names and the practical application of that knowledge.

Knowledge of the structure of identifier names has practical applications in source code comprehension and software development and maintenance. Analysis of C function names by Caprile and Tonella has been applied to automate the refactoring of names [7]. Høst and Østvold undertook detailed

analysis of Java method names [8] and found relationships between method names and method implementation in terms of the micro-patterns [11] found in the compiled bytecode. This knowledge was then applied to develop the automated detection of method naming errors and recommendation of candidate refactorings [9].

Singer and Kirkam [12] identified a link between Java class names and the micro-patterns found in the implementation using the approximation that Java class names are of the form JJ^*NN^+, where JJ represents an adjective and NN a noun[1]. However, the link was based on the assumption that the rightmost noun is an indicator of the class's implementation, and no detailed analysis of the class identifier names was undertaken.

The structure of Java class identifier names was investigated as part of a study of the cognitive aspects of identifier names as a form of communication [6]. The investigation found developers used a variety of morphological and grammatical artifices when constructing identifier names, many of which are not proposed by programming conventions. However, the investigation was conducted using source code from multiple programming languages, not Java alone, and was not a comprehensive survey of naming practice.

Deißenböck and Pizka [13] proposed a scheme of concise and consistent naming where a single identifier name represents a single concept throughout the program, and that identifier names are composed so as to represent discrete concepts unambiguously. In a follow up experiment, Lawrie *et al.* [14] surveyed the identifier names found in 48 MSLOC of C, C++, Fortran and Java source code to determine the extent of violations of concise and consistent naming. The syntactic methodology employed by Lawrie *et al.* identifies potential violations of concise and consistent naming which include some conventional patterns of naming found in Java inheritance trees.

Identifier naming conventions were used by Abebe *et al.* [15] to identify *smells* in source code. The smells are predicated on deviations from suggested identifier naming conventions that arise from programming conventions, and, to a lesser extent, deviation from established conventions arising from identifier naming praxis. A single rule concerns the grammatical structure of class identifier names, and states that class identifier names should contain at least one noun and not contain any verbs. However, this work is based on naming conventions that are guidelines of good practice, rather than a knowledge of naming conventions found in practice.

Previous research has investigated identifier naming in Java and detailed investigations have been constrained to method identifier names. Despite the discovery of practical applications for a comprehensive, detailed understanding of the structure of identifier names, there has been no detailed investigation of Java class identifier names.

[1]Throughout this paper we use the Penn Treebank PoS tag set ftp://ftp.cis.upenn.edu/pub/treebank/doc/tagguide.ps.gz

III. Methodology

From a database of 623,000 identifier names extracted from 16.5 MSLOC[2] of Java source code found in 60 open source Java projects[3], we extracted and tokenised 120,000 unique class identifier names using our source code mining tool [16], [17]. The tool was configured to detect and ignore both test code and generated source code, using heuristics. We applied two separate analytical techniques to each identifier name: part-of-speech tagging to help identify any common grammatical patterns, and an investigation of the origins of class identifier name components, if any, found within the names of the immediate super class and implemented interfaces.

We also undertook a case study of Freemind[4], a Java mind-mapping application, to establish whether unconventionally structured class identifier names indicate that either there is a problem with the class name – so that it can be refactored to a more conventionally structured name – or that there is a problem with the class itself. And, in the latter case, whether any possible refactorings could be identified that might result in two or more conventionally named classes.

A. Analysing Grammatical Composition

Previous investigations of C function names and Java method names [18], [8], [19] used part of speech (PoS) tagging to identify grammatical patterns in names. Some teams have created their own PoS tagger tuned to identifier names [8], while others have used an off-the-shelf PoS tagger [19]. We followed the latter route by using the Stanford Log-linear PoS tagger[5].

Our initial experiments were undertaken with the default tagger provided with the Stanford PoS tagger. Using the tagger, which is trained on a corpus of articles taken from The Wall Street Journal, we observed, through manual inspection, error rates of $15-30\%$ for whole identifier names, depending on the project analysed. The chief sources of error appeared to be the difference between the structure of Java class names and the conventional English sentences that form the tagger's training corpus, and the presence of abbreviations and a technical vocabulary. The consequence of this was that the tagger was trying to tag unknown words in an unrecognised context. As a result we saw common English words tagged as foreign words. We also observed issues related to the resolution of ambiguous PoS. For example the class name `ContentHandler` consisting of two nouns was consistently tagged as an adjective followed by a noun, and the word 'set' was often tagged as a verb when used as a noun.

We experimented by creating more sentence-like structures from class names by including the Java keywords from the class declaration and appending the name of the super class and implemented interface. For example, for the class `CustomPropertiesTagHandler`, found in GanttProject,

[2]Sloccount http://www.dwheeler.com/sloccount/

[3]Full information available at http://www.facetus.org.uk/corpus.html

[4]v0.9.0RC9 http://freemind.sourceforge.net/

[5]http://nlp.stanford.edu/software/tagger.shtml

which has no explicit super class and implements two interfaces, we would construct the phrase *class custom properties tag handler implements tag handler and parsing listener*. We saw no significant improvement in accuracy.

We decided to train our own PoS tagger using the Stanford PoS tagger. We extracted a training corpus of 9,000 class names at random from 13 of the 60 projects analysed, including the Java library. Each name was tagged manually and any class names that could not be unambiguously tagged were discarded. A tagger was trained on the corpus. A separate test corpus of 2,000 class identifier names was created by manually tagging class names extracted randomly from a further 8 of the projects analysed. An accuracy of 95% was achieved when tagging individual words and abbreviations against the test corpus. However, unrecognised words and abbreviations were only tagged with 83% accuracy, resulting in an accuracy rate for whole identifier names of 87% being reported by the tagger in test mode.

Whilst not perfect, the accuracy is an improvement on the default tagger. Høst and Østvold [8] state the accuracy of their PoS tagger is better than 97% as the result of manual inspection, but are unclear whether this figure is for individual words or for whole method identifier names. Falleri *et al.* [19] claim a PoS tagging accuracy of 96% for TreeTagger's default tagger for identifier names, but, again, are unclear whether this relates to individual tags, or whole identifier names.

B. Inheritance analysis

The common grammatical structures found in class identifier names do not identify how developers encode information in class names. Class names in some inheritance hierarchies in the Java library repeat part of the super class name. For example the class `HTMLEditorKit`, found in the `javax.swing.text.html` package, is a subclass of `StyledEditorKit`. Similarly the collections classes in the `java.util` package often follow a pattern of naming where a base interface name is retained through intermediate classes to the various implementations. Taking the `List` classes as an example, the `List` interface extends the `Collection` interface, and is implemented in a class by `AbstractList`. The common list classes – e.g. `ArrayList` and `LinkedList` – then extend `AbstractList`. In other words, a basic implementation of a core interface is often named `Abstract`$\langle interface\ name \rangle$, and specialised implementations extend the abstract class and replace `Abstract` with an adjectival phrase describing the implementation.

We analyse the incorporation of component words in a class identifier name from the immediate super class and any implemented interfaces. Class names were partitioned into six groups according to whether the class explicitly extends a super class – Java classes that do not explicitly declare a super class extend the root class `Object` – and the number of interfaces implemented. The notation we use consists of the letters E for extends and I for implements with each letter being followed by a subscript indicating the number of super classes extended or interfaces implemented: the values of the subscript being 0, 1 and n where the last means two or more and can only be a subscript to I. For example a class that extends a super class and implements no interfaces is classified as $E_1 I_0$.

In this study we investigate lexical inheritance, that is we investigate whether component words from the super class or implemented interface names are found in the class identifier name. Accordingly we draw no distinction between identically named super classes from different packages and ignore any package name that might have been specified by the developers in the *extends* and *implements* clauses of class declarations. Similarly we ignore generic type names where they are specified.

C. Case study

Freemind is a mind-mapping application written in Java with an 11 year development history. The application consists of a GUI that allows the user to edit, format and annotate a treelike-graph structure of text nodes. The mind maps are stored in an XML format, and can be exported to a range of external formats including HTML and OpenOffice Writer, and as images, flash animations and Java applets.

Using the results of the grammatical and the inheritance analyses, we identified those classes with identifier names that do not conform to the commonly occurring grammatical patterns found in Freemind class names. The subset of classes was then inspected to determine whether the class might be named according to one of the more commonly occurring grammatical patterns, or, if the class appeared to be appropriately named, whether the unusual name might be indicative of a potential refactoring of the class into two or more classes with more conventional names.

IV. Results

To present our results, we combine the individual Treebank adjective, noun and verb PoS tags so that JJ, NN and VB include all forms of adjective, noun, and verb respectively. By collapsing the Penn Treebank categories we create a simplified tagset similar to that used by Høst and Østvold [9].

A. Grammatical Structure

Table I shows the absolute and relative frequencies for the most common grammatical patterns found in 120,000 class identifier names extracted from 60 open source Java projects. The four grammatical forms given in the table comprise 90% of the identifier names analysed. The remaining 10% of class identifier names are formed using a variety of patterns, some with only a single instance. Of note is that Singer and Kirkham's approximation of Java class identifier names, JJ^*NN^+ [12], can be realised by merging the two most common categories and includes 85% of the class identifier names analysed.

B. The influence of inheritance

Table II shows the distribution of classes according to type of inheritance in classes found in all 60 projects. In total

TABLE I
COMMON PART OF SPEECH PATTERNS AND FREQUENCIES FOR ALL PROJECTS

Pattern	Absolute Frequency	Relative Frequency
NN^+	88489	0.73
JJ^+NN^+	14833	0.12
$NN^+JJ^+NN^+$	3579	0.03
$VB\ NN^+$	2918	0.02

80% of classes are related to another class by inheritance. Some 58% of classes extend a super class, with the majority, 45% in the E_1I_0 category, not implementing an interface. Overall, 35% of classes implement one or more interfaces, and, reflecting the situation with class-based inheritance, the majority of classes implementing an interface do not also extend a class. Indeed only 13% of classes, overall, take advantage of both dimensions of inheritance available in Java.

TABLE II
DISTRIBUTION OF INHERITANCE CATEGORIES FOR ALL PROJECTS

Category	Absolute Frequency	Relative Frequency
E_0I_0	25056	0.21
E_0I_1	22000	0.18
E_0I_n	4280	0.04
E_1I_0	54232	0.45
E_1I_1	11668	0.10
E_1I_n	3350	0.03

Figure 1 shows the variation in the proportions of classes in the inheritance categories for the 60 projects investigated. There is considerable variation between projects indicating different teams' preference for particular types of inheritance. For example, Egantt and Javacc have no classes that implement more than one interface, while, at the other end of the scale, 48% of JFreeChart classes do. By far the most marked variation is in the proportion of classes that extend a single super class (E_1I_0) ranging from 70% of classes for ArgoUML to 15% for Tapestry, where 77% of classes are found in the E_0I_0 and E_0I_1 categories.

The distribution of the 4 dominant grammatical patterns in the 6 inheritance categories is shown in Table III. Of note are the similar frequencies of the grammatical forms for the 5 categories where inheritance is involved, and that the E_0I_0 category has a noticeably greater proportion of identifier names composed exclusively of nouns.

TABLE III
RELATIVE FREQUENCY OF MOST COMMON GRAMMAR PATTERNS BY INHERITANCE CATEGORY

	NN^+	JJ^+NN^+	$NN^+JJ^+NN^+$	VB^+NN^+
E_0I_0	0.85	0.08	0.01	0.01
E_0I_1	0.73	0.15	0.02	0.02
E_0I_n	0.75	0.15	0.03	0.01
E_1I_0	0.68	0.12	0.04	0.03
E_1I_1	0.70	0.15	0.04	0.02
E_1I_n	0.75	0.14	0.04	0.02

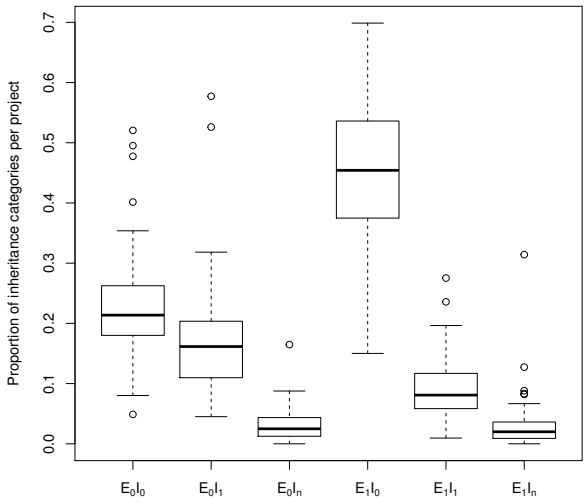

Fig. 1. Distribution of inheritance categories in 60 Java projects

Table IV shows the frequency with which elements of super class and interface names are repeated in class identifier names for the different inheritance categories (the E_0I_0 category is omitted from the table). The columns headed *all* contain the frequency for each category with which the super class or interface name is repeated in its entirety and uninterrupted. The columns headed *fragment* give the frequency with which one or more fragments of the super class or interface name are repeated. The *both* column shows the frequency with which elements from both sources are repeated in the class identifier name. For each inheritance category approximately 80% of class identifier names incorporate elements from either the super class or implemented interface identifier names. The repetition of fragments of super class identifier names is more common than the repetition of the entire super class name. Repetition of interface identifier names and fragments occurs with a similar frequency, apart from the E_1I_n category where fragments of interface names are repeated more often than the entire interface name. Where both dimensions of inheritance are used – the E_1I_1 and E_1I_n categories – the super class name is the more common source of class identifier name components. Indeed some 40% of identifier names in the E_1I_1 category and 43% in the E_1I_n category repeat component words from only the super class name, compared to 18% and 22%, respectively, that repeat component words from interface identifier names alone. We also found that 8% of class identifier names in the E_0I_n category and 1% in the E_1I_n category incorporate elements from two or more implemented interface identifier names.

The most common grammatical forms of class identifier names with component words inherited from a super class or interface are given in Table V for each category. In this table we introduce a notation for tagging super class and interface names, as well as their fragments. The tags are SC to represent a super class and SCF for a fragment of the super class name. Similarly we use II and IIF for interface names and fragments of interface names respectively.

TABLE IV
RELATIVE FREQUENCY DISTRIBUTION OF NAME INHERITANCE WITHIN INHERITANCE CATEGORIES FOR ALL PROJECTS

Category	Super Class Name		Interface Name		Both
	All	Fragment	All	Fragment	
E_0I_1	-	-	0.39	0.37	-
E_0I_n	-	-	0.38	0.40	-
E_1I_0	0.23	0.58	-	-	-
E_1I_1	0.14	0.53	0.24	0.21	0.27
E_1I_n	0.11	0.50	0.15	0.25	0.18

TABLE V
COMMON GRAMMATICAL FORMS OF CLASS NAME COMPONENT INHERITANCE

Category	Grammatical Form	Relative Frequency
E_0I_1	NN^+II	0.19
	NN^+IIF^+	0.17
	$II\ NN^+$	0.09
	IIF^+NN^+	0.05
E_0I_n	NN^+II^+	0.17
	NN^+IIF^+	0.15
	II^+NN^+	0.09
	IIF^+	0.07
	IIF^+NN^+	0.06
E_1I_0	NN^+SCF^+	0.30
	NN^+SC	0.13
	$SCF^+NN^+SCF^+$	0.05
	SCF^+NN^+	0.05
	$JJ^+NN^+SCF^+$	0.03
	JJ^+SC	0.03
E_1I_1	NN^+SCF^+	0.15
	$II\ SCF^+$	0.11
	NN^+SC	0.06
	SCF^+IIF^+	0.04
	IIF^+	0.04
	SCF^+II	0.03
	NN^+IIF^+	0.03
E_1I_n	NN^+SCF^+	0.20
	NN^+SC	0.05
	IIF^+	0.05
	NN^+IIF^+	0.04
	IIF^+NN^+	0.04
	SCF^+NN^+	0.03

For each category, the most common patterns incorporate the inherited name elements as a suffix. This is the pattern found in our earlier example from the `java.swing.text.html` package where `HTMLEditorKit` is a subclass of `StyledEditorKit`. In other words the part of the super class name that defines what the class is is retained and a word or words are added as a prefix to create the specialised version of the super class. Similarly where part of an interface name is retained as a suffix, the interface name defines the type of thing the class is. For example, the class `GlobPatternMapper` found in Ant implements the `FileNameMapper` interface.

The patterns where a fragment of the super class or interface name is used as a prefix for the class name are less common. An example is the `InstructionHandle` class in MultiJava, which extends the super class `AbstractInstructionAccessor`. In this case the focus of the class's activity, an instruction, is the discriminating term that is perpetuated through inheritance.

The $SCF^+NN^+SCF^+$ pattern found in the E_1I_0 and E_1I_1 categories, arises in class identifier names like `BuddyPluginPasswordException` found in Vuze which has the super class `BuddyPluginException`, where one or more nouns are inserted into the superclass name to indicate the specialisation.

Where both dimensions of inheritance are used, the E_1I_1 and E_1I_n categories, the most common name inheritance components are NN^+SCF^+, $II\ SCF^+$ and NN^+SC. The E_1I_1 category contains identifier names composed exclusively of words repeated from both the super class and implemented interface identifier names. An example of the $II\ SCF^+$ pattern is the JabRef class `FieldTextArea` which is a sub class of `JTextArea` and implements the `FieldEditor` interface.

The repetition of entire interface names as suffixes, the NN^+II pattern, is absent from the E_1I_n category despite being the most common pattern in both the E_0I_1 and E_0I_n categories. An example of the class name pattern is the `LinkedHashMap` class in the `java.util` package which has the super class `HashMap`. Many of the instances of the $IINN^+$ pattern are examples of class identifier names ending in `Impl`, an abbreviation for implementation, such as `GroovyFeatureImpl` found in NetBeans. Also relatively common are class identifier names composed exclusively of fragments of an interface name, i.e. the pattern IIF^+. This can result from the use by some developers of the letter `I` as a prefix to indicate an interface identifier name, e.g. `IDialogSettings` found in Eclipse, where the implementing class is named `DialogSettings`. Other examples of the IIF^+ pattern include the class `JNDIResource`, found in JBoss, that implements the interface `JNDIResourceMBean`, and the Java library class `RMISocketFactory`, which implements the interfaces `RMIClientSocketFactory` and `RMIServerSocketFactory`.

C. Freemind

We extracted 652 class identifier names from Freemind and the frequency of the most common grammatical patterns can be seen in Table VI. The four most common patterns are the same as those found with the greatest frequency for all projects (Table I), and were found with similar frequencies apart from the $VBNN^+$ pattern where the frequency for Freemind was almost double that for all the projects analysed. In total the four most common grammatical patterns describe 91% of the class identifier names in Freemind.

The distribution of class identifier names in the inheritance categories for Freemind given in Table VII show a greater proportion of classes in Freemind either extend a super class or implement one or more interfaces than is observed across all the projects analysed (see Table II). Indeed, 66% of classes in Freemind extend a super class, as opposed 58% across all classes.

TABLE VI
COMMON PART OF SPEECH PATTERNS AND FREQUENCIES FOR FREEMIND

Pattern	Absolute Frequency	Relative Frequency
NN^+	459	0.70
JJ^+NN^+	81	0.12
$VB\,NN^+$	39	0.06
$NN^+JJ^+NN^+$	20	0.03

Table VIII shows the distribution of common grammatical forms of class identifier names that include words derived from the super class or implemented interfaces for each of the inheritance categories in Freemind. While many patterns are common to the same categories in Table VIII and Table V, there are key differences that indicate the existence of project-specific naming conventions in Freemind. For example, the two most frequently occurring patterns in the E_0I_1 and E_0I_n categories overall, NN^+II^+ and NN^+IIF^+, are much more common in Freemind. Furthermore, the pattern NN^+IIF^+, where a fragment of the interface name is repeated in the class identifier name, is the more frequent of the two in Freemind and accounts for 29% of the class identifier names in the E_0I_n category.

TABLE VII
DISTRIBUTION OF INHERITANCE CATEGORIES FOR FREEMIND

Category	Absolute Frequency	Relative Frequency
E_0I_0	82	0.13
E_0I_1	128	0.20
E_0I_n	14	0.02
E_1I_0	318	0.49
E_1I_1	78	0.12
E_1I_n	32	0.05

The $NN^+SCF^+NN^+$ pattern observed with a frequency of 7% in the E_1I_0 inheritance category in Freemind occurs with much lower frequency across the projects analysed. Inspection of the instances of the $NN^+SCF^+NN^+$ pattern reveals a local naming convention, an example of which is the class `MindMapCloudModel` that implements the interface `CloudAdapter`. The pattern occurs mainly in a small cluster of packages where it is used for classes that form fundamental components of Freemind's mind maps.

Another feature of Freemind is the relative prominence of class names with grammatical patterns beginning with a verb in the E_1I_0 and E_1I_1 categories. Identifier names of the $VB\,NN^+SCF^+$ pattern are mainly of the form $VB\,NN^+$, when the origin of component words is ignored, and, as already noted, Freemind has a relatively high frequency of identifier names of this general pattern (See Table VI). As we discuss below, many of the classes responsible for handling user initiated actions in the GUI start with a verb. The $VB\,NN^+SCF^+$ pattern is also prominent in the E_1I_1 category, where it is notable that the NN^+SCF^+ pattern occurs with a much higher frequency in Freemind (38%) than amongst all the projects analysed (15%). The IIF^+ pattern found in the E_1I_1 and E_1I_n categories for all the

TABLE VIII
COMMON GRAMMATICAL FORMS OF CLASS NAME COMPONENT INHERITANCE FOR FREEMIND

Category	Grammatical Form	Relative Frequency
E_0I_1	NN^+IIF^+	0.27
	NN^+II	0.20
	IIF^+NN^+	0.05
	$JJ^+NN^+IIF^+$	0.05
	$II\,NN^+$	0.04
E_0I_n	NN^+IIF^+	0.29
	NN^+II	0.21
	IIF^+	0.07
E_1I_0	NN^+SCF^+	0.35
	$NN^+SCF^+NN^+$	0.07
	NN^+SC	0.06
	SCF^+NN^+	0.05
	$JJ^+NN^+SCF^+$	0.04
	$VB\,NN^+SCF^+$	0.04
	$JJ\,SCF^+$	0.04
E_1I_1	NN^+SCF^+	0.38
	IIF^+SCF^+	0.09
	$II\,SCF^+$	0.08
	$VB\,NN^+SCF^+$	0.05
	SCF^+NN^+	0.05
E_1I_n	NN^+SCF^+	0.28
	SCF^+NN^+	0.09

projects surveyed are not found in the same categories for Freemind. Where components from implemented interface identifier names are incorporated into class names in the E_1I_1 and E_1I_n categories in Freemind so are fragments of the super class identifier name.

In addition to the four most common grammatical patterns given in Table VI, further patterns were identified accounting for a total of 53 class names (see Table IX), or a relative frequency of 0.08. The most common grammatical pattern amongst this group was $VB\,NN^+IN\,NN^+$, where IN represents a preposition. Each of the 53 unconventionally named classes was inspected to understand whether the name used was appropriate and a clear reflection of the role the class plays in the application. If the name did not meet those criteria we explored possible name refactorings that adhered to the established naming conventions found in the project and were permitted within the same namespace. We also considered whether the class might be refactored into classes that could be named conventionally.

The majority of the 53 classes inspected represented actions taken by the user in the GUI, or coordinating events such as automatically saving all open files. On the whole these class identifier names clearly described the role of each class and were not prone to name refactoring.

Some class names were identified that could be refactored to more conventional identifier names. Three were member classes with identifier names prefixed with `My`, which gives the reader little information about the origins of the class, or the detail of the functionality they might expect to encounter. For example the class `MyRenderer`, a member class of the class `ImportAttributesDialog` in the

package freemind.modes.mindmapmode.attributeactors, is responsible for rendering the cells of a tree used for display in the dialogue. Other member classes of the same class have clear identifier names reflecting the detail of their purpose, e.g. AttributeTreeNodeInfo. To make the class name consistent with the other member classes, and to improve clarity, we suggest the name is refactored to AttributeTreeCellRenderer, which adheres to the common NN^+ pattern.

The class ArrayListTransferable is designed to protect an array list from modification while it is transferred between two objects. The adjective has been placed after the noun phrase it is intended to modify, and we suggest the name is refactored to TransferableArrayList, which is both clearer and conforms to the common JJ^+NN^+ pattern.

The class ThreeCheckBoxProperty is a GUI component that implements a button used in dialogues where the user has a number of settings available. The button cycles through three states when clicked, which are represented by a plus sign, a minus sign and an empty box and have the meanings change property, remove property and ignore, respectively. The GUI component does not look like a check box or behave like one, so that aspect of the name appears to be incorrect. It is used in properties dialogues, but *property* in isolation does not adequately represent its usage context. The common name for this type of widget is a *tri-state checkbox*, so the most appropriate name refactorings are TristateCheckBox or TristateButton, both of which conform to the JJ^+NN^+ pattern, and do not contain extraneous detail about the type of dialogue in which the component is used.

We also found one instance of a spelling mistake: a class named FileChooseListener, which should have been FileChooserListener to be consistent with the name of the Swing JFileChooser instance it creates. The spelling mistake was identified as the result of the PoS tagger recording Choose as a verb, thus giving the identifier name the pattern NN^+VBNN^+, which is relatively uncommon. The remaining four instances of this pattern in Freemind are a part of a local naming convention in the freemind.controller.filter.condition package. Most classes extend NodeCondition or one of its subclasses and follow a consistent naming scheme based on their position in the hierarchy. A feature of the naming scheme is the insertion of a verb between the two component words of the super class. The classes are used to support a filtering mechanism that selects particular nodes for display. For example the class NodeContainsCondition is used to test whether a node contains a particular condition or attribute. We consider the class identifier names not to be a problem because the unconventional naming is used consistently, and the classes, on inspection, appear to function as described. However, we would argue that the awkward nature of the naming pattern may be a design smell, and that a more conventional design, e.g. the visitor pattern [20], should result in more conventional class identifier names.

The class StdOutErrLevel is a candidate for refactoring. StdOutErrLevel is a member class of freemind.main.StdFormatter used in logging and was identified because the intended abbreviations of 'output' as 'out' and 'error' as 'err' are also words. On initial reading of the name, it appears to be an abbreviation of *standard output error level*, or *stdout error level*. On inspection, the StdFormatter class is responsible for formatting log records for Freemind, and StdOutErrLevel is used to assign the logging threshold for the *stdout* and *stderr* streams. The developers combine the task of setting the logging thresholds for two streams into the same object, before specifying each output stream in the call to the constructor (see Figure 2). There are two solutions: either the class name is refactored to remove the reference to both standard streams, or the class, which wraps the java.util.logging.Level class without adding any functionality, is removed and replaced by direct calls to the Level library class. The latter is the preferable solution as it results in a less cluttered class that is easy to read.

```
/**
 * Level for STDOUT activity.
 */
 final static Level STDOUT =
    new StdOutErrLevel("STDOUT", Level.WARNING.
        intValue()+53);

 /**
  * Level for STDERR activity
  */
  final static Level STDERR =
    new StdOutErrLevel("STDERR", Level.SEVERE.
        intValue()+53);
```

Fig. 2. Partial listing from freemind.main.StdFormatter

EdgeWidthBackTransformer is one of a group of member classes of StylePatternFrame in the package freemind.modes.mindmapmode.dialogs that transform strings to widths and back again. The class performs the inverse function of the class EdgeWidthTransformer and invokes the method transformStringToWidth. We suggest the renaming the class to StringToEdgeWidthTransformer may be clearer and more consistent. However, the identifier name remains unconventional.

D. Threats to Validity

As with any empirical study there are threats to validity. In this case threats to validity concern construct and external validity. We do not consider internal validity because we make no claims of causality. Also we have not used any statistical tests, so we do not consider statistical conclusion validity.

a) Construct Validity: The key threat to construct validity is the accuracy of the PoS tagger used in the experiment. The Stanford PoS tagger's test mode for our tagger reports an accuracy of 95% for individual words and an accuracy of 85% for whole identifier names. The sources of error include words that commonly have more than one part of speech, abbreviations and unknown words. However, despite the error

TABLE IX
CLASSES INSPECTED IN FREEMIND

Package	Class name	Refactor Name	Refactor Class	Comment
accessories.plugins	ExportToImage	No	No	Describes action initiated in UI
	ExportToOoWriter	No	No	Describes action initiated in UI
	ExportWithXSLT	No	No	Describes action initiated in UI
	FitToPage	No	No	Describes action initiated in UI
	JumpToMapAction	No	No	Describes action initiated in UI
	MyFreemindPropertyListener	Yes	No	Rename AutomaticLayoutPropertyListener
	SaveAll	No	No	Describes action initiated in UI
	UnfoldAll	No	No	Describes action initiated in UI
accessories.plugins.dialogs	ArrayListTransferable	Yes	No	Rename TransferableArrayList
accessories.plugins.time	ReplaceAllInfo	No	No	Local naming convention
	ReplaceSelectedInfo	No	No	Local naming convention
accessories.plugins.util.xslt	FileChooseListener	Yes	No	Rename FileChooserListener
freemind.common	ThreeCheckBoxProperty	Yes	No	Rename TristateButton
freemind.controller	AboutAction	No	No	Describes action initiated in UI
	DisposeOnClose	No	No	Wraps Swing GUI action
	HideAllAttributesAction	No	No	Describes action initiated in UI
	MoveToRootAction	No	No	Describes action initiated in UI
	ShowAllAttributesAction	No	No	Describes action initiated in UI
	ShowSelectionAsRectangle	No	No	Describes action initiated in UI
	ZoomInAction	No	No	Describes action initiated in UI
	ZoomOutAction	No	No	Describes action initiated in UI
freemind.controller.filter	CreateNotSatisfiedConditionAction	No	No	Local naming convention
freemind.controller.filter.condition	AttributeCompareCondition	No	No	Local naming convention
	AttributeExistsCondition	No	No	Local naming convention
	AttributeNotExistsCondition	No	No	Local naming convention
	ConditionNotSatisfiedDecorator	No	No	Local naming convention
	IgnoreCaseNodeContainsCondition	No	No	Local naming convention
	NodeCompareCondition	No	No	Local naming convention
	NodeContainsCondition	No	No	Local naming convention
	NoFilteringCondition	No	No	Local naming convention
freemind.extensions	AllDestinationNodesGetter	No	No	Describes action initiated in UI
freemind.main	StdOutErrLevel	No	Yes	Remove member class
freemind.modes	NodeDownAction	No	No	Describes action initiated in UI
	SaveAsAction	No	No	Describes action initiated in UI
freemind.nodes.common	CommonToggleFoldedAction	No	No	Describes action initiated in UI
freemind.modes.mindmapmode	DoAutomaticSave	No	No	Time-based backup task
	ExportBranchToHTMLAction	No	No	Describes action initiated in UI
	ExportToHTMLAction	No	No	Describes action initiated in UI
	SetImageByFileChooserAction	No	No	Describes action initiated in UI
	SetLinkByFileChooserAction	No	No	Describes action initiated in UI
freemind.modes.mindmapmode.actions	AddLocalLinkAction	No	No	Describes action initiated in UI
	ChangeArrowsInArrowLinkAction	No	No	Describes action initiated in UI
	NodeUpAction	No	No	Describes action initiated in UI
	RemoveAllIconsAction	No	No	Describes action initiated in UI
	SelectAllAction	No	No	Describes action initiated in UI
	SetLinkByTextFieldAction	No	No	Describes action initiated in UI
	UsePlainTextAction	No	No	Describes action initiated in UI
freemind.modes.mindmapmode.attributeactors	MyRenderer	Yes	No	Rename AttributeTreeCellRenderer
	ToggleAllAction	No	No	Describes action initiated in UI
freemind.modes.mindmapmode.dialogs	EdgeWidthBackTransformer	Yes	No	Rename StringToEdgeWidthTransformer
freemind.modes.viewmodes	CommonToggleChildrenFoldedAction	No	No	Describes action initiated in UI
freemind.view.mindmapview	Selected	Yes	No	Rename SelectedNodes
freemind.view.mindmapview.attributeview	MyFocusListener	Yes	No	Rename AttributeTableFocusListener

rate we successfully identified unconventionally constructed identifier names that were candidates for refactoring as well as a possible design smell. An associated threat is that the training and testing corpora were hand tagged by the first author, a native speaker of English, which is a possible source of bias.

The consequence of collapsing the Penn Treebank PoS tags is that 50 proper nouns and 15049 plural nouns are included in the NN tag, and a total of 1393 verb forms are included in the VB tag. These account for 13% of class identifier names containing nouns and 24% of those containing verbs. Only 0.2% of adjectives are tagged as JJR and JJS. While this approach hides some detail, it allows the observation of a general form of class identifier name containing a verb, which might otherwise have been missed.

b) External Validity: We used a corpus of 60 open source Java projects totalling some 16.5 MSLOC as the source of the class identifier names analysed in this project. The

60 projects are drawn from a variety of domains to reduce the influence of domain specific identifier names or naming styles. We found considerable variation in the proportions of class and identifier based inheritance used in the projects. Accordingly, while our observations of the patterns of the reuse of component words from the names of super classes and implemented interfaces are reliable for the corpus analysed, caution should be exercised when extrapolating the proportions of these patterns to other projects.

V. Discussion

We employed two methods of analysing of class identifier name structure. The first relied solely on the parts-of-speech used in the class names, and the second considered the origins of the component words in the names of the super class and implemented interfaces. We found that more than 90% of class identifier names can be described using four simple grammatical patterns. Despite the advice given in programming conventions [4], [3] that class identifier names should be nouns, we found that a proportion – around 2% – incorporate verbs and describe actions, rather than entities. From inspection of a sample of the class identifier names containing verbs, we found that many were related to actions initiated by the user in GUI environments.

Programming conventions offer no advice on whether, or how to incorporate information from super class or interface names in class identifier names. Praxis, in the Java library, for example, is for some class and interface hierarchies to retain part of the class identifier name through the inheritance hierarchy. Our analysis of the origins of component words in class identifier names found that 70-80% of classes that extend a superclass or implement an interface include one or more words repeated from the super class or implemented interface name (See Table IV, Table V and Table VIII).

In general, class identifier names repeat fragments of the super class or interface name, rather than the entire name, and it is words found in the super class names that are repeated with the higher frequency. Manual inspection showed that fragments are mostly derived from the latter, or right hand, part of the super class or interface name. Though a common pattern found in exception classes involves the insertion of words in the super class name. We were unable to identify any obvious mechanisms from single generation inheritance that explain how a decision is made to repeat either part or all of the super class or interface name. We expect that it will be possible to derive heuristics from inheritance trees where name fragments are repeated over more than one generation.

The repetition of components of super class and implemented interface identifier names in some class identifier names appears to violate the conciseness rule of Deißenböck and Pizka's system of concise and consistent naming [13]. A *concise* name is one that unambiguously represents a given concept within a program. For example, the identifier names `position` and `absolutePosition` would break the conciseness rule because the concept of 'position' *contains* the concept 'absolute position'.

Lawrie *et al.* [14] used a syntactic methodology to investigate violations of concise and consistent naming. They defined *Type I* and *Type II* syntactic violations, both of which imply that the conciseness rule has been broken, and may indicate a failure of the consistency rule. A Type I violation occurs when an identifier name is repeated entirely within another identifier name; and a Type II violation occurs when an identifier name is similarly contained by two or more others. Type I violations occur in a single generation of inheritance where a class identifier name includes the *whole* of the name of either the super class or an implemented interface.

In Freemind, for example, 89 (14%) of the class identifier names we surveyed are Type I violations. Some class names appear to be genuine violations of the conciseness rule, but most are part of the process of creating program concepts through inheritance. The same is true of the Type II violations we identified. In the latter case the false positives are typified by classes that implement a common interface or base class, e.g. `SortedComboBoxModel` and `ClonedComboBoxModel` both implement the Swing interface `ComboBoxModel`, and describe the concept hierarchy clearly. Lawrie *et al.* suggested that parts-of-speech may help discriminate between identifier names that represent new concepts and those that are genuine violations. The adjectives in our example fulfil that role, but further study is required to confirm the viability of the method.

Importantly, around 20-30% of class identifier names in each inheritance category were found not to incorporate any words derived from the identifier names of the extended super class or implemented interfaces. Further investigation is needed to identify the occasions on which names are incorporated and those when they are not. One approach is to derive rules of name inheritance from existing behaviour within a code base, which may have a practical application by alerting a developer to an unusual repetition of a component word in a class identifier name, or the omission of a component that is commonly repeated. For example, the Java library interface names `Cloneable` and `Serializable` are rarely incorporated in the identifier names of implementing classes. While such a solution is attractive, it does not explain when and why super class and interface name components are repeated. A more detailed approach, analysing identifier names in terms of their grammatical structure, their semantics, and their role or position in the inheritance hierarchy, may result in methods that predict the circumstances under which component words are repeated and which words should be repeated.

The common class identifier naming patterns are familiar mechanisms developers use to communicate ideas. Høst and Østvold demonstrated that the link between Java method identifier names is sufficiently strong that poor quality or misleading names can be identified and candidate refactorings suggested [9]. By identifying candidates for renaming, such as `ThreeCheckBoxProperty`, and a possible design smell, we show that practical results may be achieved through the recognition of unconventionally constructed Java class identifier names. However, while identifier names may reflect

the implementation of a class, conventionally structured class identifier names are not a guarantee of flawless design.

Knowledge of the common conventional grammatical patterns of class identifier naming can be incorporated in IDE-based tools to support the creation of class identifier names that are more readily understood by developers, and that conform to project standards determined by software project managers. Such a tool could alert the developer to an unconventionally structured name and, eventually, recommend possible improvements, including the incorporation of words used in the super class and interfaces, if any. Developers new to a project would have a ready made style guide to support the creation of class identifier names that are familiar to existing colleagues. The same knowledge can be leveraged to provide quality assurance for software project managers that is an improvement on the functionality of current tools. For example, CheckStyle[6] provides only very basic checks of the typographical structure of identifier name, e.g. whether a class identifier name begins with an upper case letter.

For software maintainers new to a project, a tool that extracts the project's class naming conventions provides an overview of how the project's developers encode information in identifier names, particularly the extent to which names reflect inheritance. Such information supports program comprehension by identifying which component words are repeated in inheritance trees and can be used to identify related classes and help target lexical searches.

VI. Conclusions

Through the analysis of class identifier names extracted from 60 Java open source projects and a case study of Freemind, we have taken a step towards a detailed understanding of Java class identifier naming conventions used in practice. This paper makes three contributions:

1) We identify the common grammatical structures of Java class identifier names found in praxis and their distributions.
2) We identify the patterns by which component words from the super class or implemented interfaces are repeated in class identifier names, and record their distributions.
3) We show, for the example of Freemind, how the detailed knowledge of project-specific uncommon class identifier naming patterns can be put to practical use to help detect poor class names and design smells and thereby improve program comprehension and design.

Our analysis can be applied in practical software engineering tools to support identifier naming by making developers aware of the naming conventions used in the project they are working on. A tool could offer guidance that supports the creation of more commonly recognisable names, such as indicating when an unconventional form is being used, or making recommendations of possible identifier names during development. The same knowledge can also be applied by software project managers to configure the developers' tools with project-specific standards, and in tools for identifier name quality assurance that are a considerable improvement on current tools that check the typographical form of names.

We propose to build on this work in two ways. The first is to seek methods of determining the relationship between a class identifier name, its inheritance hierarchy and the identifier names of class members including fields and methods, to support a finer-grained analysis of the correctness, or otherwise, of class identifier names. The second is to apply the knowledge acquired in this research to develop methods for the semantic analysis of identifier names in order to support the creation of ontologies of source code.

[6]http://checkstyle.sourceforge.net/

References

[1] V. Rajlich and N. Wilde, "The role of concepts in program comprehension," in *Proc. 10th Int'l Workshop on Program Comprehension*. IEEE, 2002, pp. 271–278.

[2] F. Deissenboeck and M. Pizka, "Concise and consistent naming," in *Proc. 13th Int'l Workshop on Program Comprehension*, 2005, pp. 97–106.

[3] A. Vermeulen, S. W. Ambler, G. Bumgardner, E. Metz, T. Misfeldt, J. Shur, and P. Thompson, *The Elements of Java Style*. Cambridge University Press, 2000.

[4] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java language specification*, 3rd ed. Addison-Wesley, 2005.

[5] K. Beck, *Implementation Patterns*. Addison–Wesley, 2008.

[6] B. Liblit, A. Begel, and E. Sweetser, "Cognitive perspectives on the role of naming in computer programs," in *Proc. 18th Annual Psychology of Programming Workshop*. Psychology of Programming Interest Group, 2006.

[7] B. Caprile and P. Tonella, "Restructuring program identifier names," in *Proc. Int'l Conf. on Software Maintenance*. IEEE, 2000, pp. 97–107.

[8] 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.

[9] ——, "Debugging method names," in *Proc. of the 23rd European Conf. on Object-Oriented Programming*. Springer-Verlag, 2009, pp. 294–317.

[10] D. Raţiu, "Intentional meaning of programs," Ph.D. dissertation, Technische Universität München, 2009.

[11] J. Y. Gil and I. Maman, "Micro patterns in Java code," in *Proc. ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*. ACM, 2005, pp. 97–116.

[12] J. Singer and C. Kirkham, "Exploiting the correspondence between micro patterns and class names," in *Int'l Working Conf. on Source Code Analysis and Manipulation*. IEEE, Sept. 2008, pp. 67–76.

[13] F. Deißenböck and M. Pizka, "Concise and consistent naming," *Software Quality Journal*, vol. 14, no. 3, pp. 261–282, Sep 2006.

[14] D. Lawrie, H. Feild, and D. Binkley, "An empirical study of rules for well-formed identifiers," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 19, no. 4, pp. 205–229, 2007.

[15] S. L. Abebe, S. Haiduc, P. Tonella, and A. Marcus, "Lexicon bad smells in software," in *Proc. Working Conf. on Reverse Engineering*. IEEE, 2009, pp. 95–99.

[16] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp, "Exploring the influence of identifier names on code quality: an empirical study," in *Proc. of the 14th European Conf. on Software Maintenance and Reengineering*. IEEE Computer Society, 2010, pp. 159–168.

[17] ——, "Improving the tokenisation of identifier names," in *ECOOP 2011, LNCS 6813*, M. Mezini, Ed. Springer-Verlag, 2011, pp. 130–154.

[18] B. Caprile and P. Tonella, "Nomen est omen: analyzing the language of function identifiers," in *Proc. Sixth Working Conf. on Reverse Engineering*. IEEE, Oct 1999, pp. 112–122.

[19] J.-R. Falleri, M. Huchard, M. Lafourcade, C. Nebut, V. Prince, and M. Dao, "Automatic extraction of a wordnet-like identifier network from software," in *18th Int'l Conf. on Program Comprehension*. IEEE, jun. 2010, pp. 4 –13.

[20] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.