Eighth IEEE International Working Conference on Source Code Analysis and Manipulation

Exploiting the Correspondence between Micro Patterns and Class Names

Jeremy Singer, Chris Kirkham
University of Manchester, UK
{jsinger,chris}@cs.man.ac.uk

Abstract

This paper argues that semantic information encoded in natural language identifiers is a largely neglected resource for program analysis. First we show that words in Java class names relate to class properties, expressed using the recently developed micro patterns language. We analyse a large corpus of Java programs to create a database that links common class name words with micro patterns. Finally we report on prototype tools integrated with the Eclipse development environment. These tools use the database to inform programmers of particular problems or optimization opportunities in their code.

1 Introduction

Software reuse is the basis of modern programmer productivity. At a high level, reuse involves *design patterns*. A design pattern is a general solution to a recurring problem. It provides a template that can be instantiated to specific cases [5]. At a low level, source code can be reused directly. Existing functionality can be packaged into libraries with a well-defined application programmer interface. Alternatively (and more grubbily) code may be cut-and-pasted from an existing program to a new program. This is the main thesis of *post-modern programming* [13]. Why write a fresh program to solve a problem, when someone else has already written the same program, or a sufficiently similar one?

This paper argues that there is another kind of software reuse, entirely distinct from the above points. *Coding conventions* should be identified and reused as much as possible. For instance, in Java, every time one sees a class name ending with `Impl`, one expects the class to be an implementation of an interface or abstract class. The implicit convention is that many, if not all, of the methods of an `Impl` class will override abstract methods from a superclass or interface.

In this paper, we consider conventions that can be expressed in terms of *micro patterns* [6]. These are single-class properties that are succinct, syntactic, non-trivial and easily verifiable. Section 2.1 explains micro patterns in more detail. Our hypothesis is that programmers implicitly expect certain classes to conform to specific micro patterns. The programmer's expectation is encoded in the name of the class. We restrict our attention to the Java programming language. A recent Gartner report [12] stated that 80% of the world's new software is being developed in managed programming languages like Java and C#. Class names are verbose and meaningful in Java programs. Names generally have a form that matches the pseudo regular expression $(adjective) \star (noun)+$. We refer to the final noun in a class name as the class name *suffix*. This is the only part of the class name that we study in this paper. Section 2.2 explains this in more detail.

This paper presents a study of Java programs to show how class name suffixes relate to micro patterns. We examine the data to infer existing informal coding conventions. It seems that certain 'types' of classes always exhibit certain micro patterns. Section 3 demonstrates this point on a wide range of real-world Java programs.

We envisage two main applications of this relationship between micro patterns and class name suffixes. We have created prototype versions of these applications, which are integrated with the Eclipse development environment [1]. The first example is an interactive popup wizard which operates at coding time. As the programmer creates new classes, the wizard (reminiscent of the late lamented Microsoft Office Paperclip) suggests properties that this class should exhibit, based on the relationship between suffixes and micro patterns as learned from previously seen programs. The wizard should be able to auto-expand the source code of the class to satisfy these properties, including annotations, modifiers and code skeletons. The second example application is a batch processing tool. This operates at code review time. It postprocesses all the classes and checks to see which micro patterns they exhibit. It should show which classes break coding conventions by highlighting violations

978-0-7695-3353-7/08 $25.00 © 2008 IEEE
DOI 10.1109/SCAM.2008.23

and suggesting alterations. Section 4 describes these tools in more detail.

We anticipate three benefits from these checking tools.

1. Maintaining the status quo. Respect of code conventions ought to reduce the risk of bugs occurring due to misunderstandings when other programmers maintain this source code at a later date. Maintenance programmers will have certain cultural expectations based on class names. For instance, a class with the suffix `Constants` would generally be expected to contain `public static final` fields.

2. The tools may highlight bugs due to bad coding practice. If a class with a suffix that conventionally exhibits some micro pattern property does not conform, then perhaps the programmer has mis-coded the class in some subtle way, and it really should conform to convention. At the very least, the programmer should be warned about his non-standard style to avoid problem (1) above.

3. Some micro patterns give opportunities for *code optimization*. If the tool can identify these micro patterns in the source code with annotations, the compiler can easily perform the appropriate transformation. It is helpful for a programmer to know that certain class properties can improve the performance of generated code. The tools may be able to provide such optimization hints. For instance, the *Data Manager* micro pattern (see Table 1 for details) is an ideal candidate for aggressive method inlining. It is helpful for the runtime compiler to know where optimizations are most likely to be beneficial. Properly targeted optimizations drive down the cost:benefit ratio, thus leading to better performance in an adaptive compilation system.

In summary, this paper makes three main contributions.

1. It presents an extensive study of the correspondence between micro patterns and class name suffixes.

2. It describes prototype versions of two tools that take advantage of this correspondence to suggest code completions or highlight convention violations. These tools are built into the Eclipse framework.

3. It provides a brief sketch of possible applications of these tools.

2 Background

2.1 Micro Patterns

Micro patterns are low-level, implementation-oriented design patterns. Rather than describing an interaction between classes, a micro pattern is a property of a single class. A class may exhibit zero or more micro patterns. Gil and Maman [6] present a catalogue of 29 micro patterns. These are intended to capture common object-oriented coding idioms. They comprise characterizations of degenerate classes, container classes and inheritance hierarchies. Each micro pattern specifies a non-trivial, formal condition on any or all of the attributes, types, name and body of a class and its components. A micro pattern specification is mechanically recognisable via simple static analysis techniques.

Consider the `Sampler` micro pattern as an example. It defines classes that have a `public` constructor and one or more `public static` fields of the same type as the class. Such classes provide clients with pre-fabricated instances of the class as well as being able to make new custom instances. In the Java API, `java.awt.Color` is a sampler class. Table 1 shows the full set of micro patterns devised by Gil and Maman [6].

2.2 Class Names

The standard Java naming convention is to merge multiple words into a single word, with each word in 'initial caps' format [15].[1] An example type name is `ByteArrayBuffer`. This type is a conjunction of three words: byte, array and buffer. The last word in a type name is known as the *suffix*. We identify the suffix by searching backward from the end of the type name string, until we reach a capital letter character or an alphanumeric character that is preceded by a separator character such as `_` or `$`. The substring from this position to the end of the string is its suffix. In the above example, the type name suffix is `Buffer`. Our hypothesis is that *type name suffix is often an indicator of micro patterns exhibited by that class*.

3 Correlation Study

In order to demonstrate that the relation between class names and micro patterns is valid in general, we must analyse a large and varied corpus of Java programs. Table 2 describes the programs we used for this study. These are all commonly available industry-standard benchmark suites and open-source Java applications, that have been used in previous research-based Java source code case studies.

The total number of classes in this corpus is 29398. The number of distinct class name suffixes is 4031. These two numbers make it clear that *suffix reuse* is common practice for Java developers.

[1] also known as 'Camel Case'.

	Micro pattern	*Definition*
Degenerate classes	Designator	Interface with no members.
	Taxonomy	Empty interface extending another interface.
	Joiner	Empty interface joining two or more superinterfaces.
	Pool	Class which declares only static final fields, but no methods.
	Function Pointer	Class with a single public instance method, but with no fields.
	Function Object	Class with a single public instance method, and at least one instance field.
	Cobol Like	Class with a single static method, but no instance members.
	Stateless	Class with no fields, other than static final ones.
	Common State	Class in which all fields are static.
	Immutable	Class with several instance fields, which are assigned exactly once, during instance construction.
	Restricted Creation	Class with no public constructors, and at least one static field of the same type as the class.
	Sampler	Class with one or more public constructors, and at least one static field of the same type as the class.
Containment	Box	Class which has exactly one, mutable, instance field.
	Compound Box	Class with exactly one non primitive instance field.
	Canopy	Class with exactly one instance field that it assigned exactly once, during instance construction.
	Record	Class in which all fields are public, no declared methods.
	Data Manager	Class where all methods are either setters or getters.
	Sink	Class whose methods do not propagate calls to any other class.
Inheritance	Outline	Class where at least two methods invoke an abstract method on `this`.
	Trait	Abstract class which has no state.
	State Machine	Interface whose methods accept no parameters.
	Pure Type	Class with only abstract methods, and no static members, and no fields.
	Augmented Type	Only abstract methods and three or more static final fields of the same type.
	Pseudo Class	Class which can be rewritten as an interface: no concrete methods, only static fields.
	Implementor	Concrete class, where all the methods override inherited abstract methods.
	Overrider	Class in which all methods override inherited, non-abstract methods.
	Extender	Class which extends the inherited protocol, without overriding any methods.
	Limited Self	Subclass that does not introduce new fields and all self method calls are to its superclass.
	Recursive	Class that has at least one field whose type is the same as that of the class.

Table 1. Gil & Maman [6] classify micro patterns as (a) Degenerate classes and interfaces which generally do not define any variable or methods, (b) Containment classes which explicitly manage their internal fields, or (c) Inheritance classes which inherit from other classes.

program	version	description
Ashes Suite	1st public release	Java compiler test programs
DaCapo	2006-10-MR2	Object-oriented benchmark suite
JEdit	4.3	Java text editor application
JHotDraw	709	Java graphics application
Jikes RVM	2.9.1	Java virtual machine, includes classpath library
JOlden	initial release	Pointer-intensive benchmark suite
JUnit	4.4	Test harness
SPECjbb	2005	Java business benchmark
SPECjvm	1998	Simple Java client benchmark suite

Table 2. Java benchmarks used in correlation study.

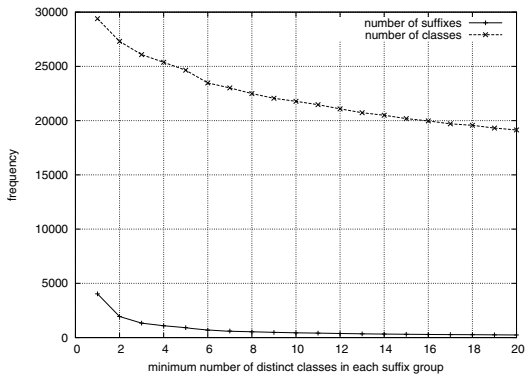

Figure 1. Graph showing how suffix groups become less significant when they must have large numbers of classes in the group.

3.1 Properties of Suffixes in the Corpus

Now we consider how the number of suffixes, and the number of classes with these suffixes, change as we filter the suffixes based on various properties.

Figure 1 shows the trend as we only consider those suffixes that have a minimum number of different classes sharing each common suffix. A point $+$ at position (x, y) on this graph indicates that there are y different suffixes in the corpus that satisfy the property that, for each one of these suffixes, there are at least x distinct classes that share that suffix. The sharply decreasing curve shows that only a few suffixes are shared across many classes. Around half the suffixes are unique to a single class. From this information, we deduce that around 50% of suffixes are *one-off*, and cannot be used to predict general class properties. Conversely, the other 50% of suffixes are *reused*, and can be used to predict class properties.

Figure 2 shows the trend as we consider only those suffixes that are dispersed across a minimum number of different Java program groups. Note that each row in Table 2 counts as a different program group. Our loose definition of a group refers to code that is developed or distributed by a single individual or organization. A point $+$ at position (x, y) on this graph indicates that there are y different suffixes in the corpus that satisfy the property that, for each one of these suffixes, there are distinct classes from at least x different program groups that share that suffix. Over 66% of suffixes occur in a single program group. There may be reuse within this program, but the reuse is not dispersed to other program groups. This may indicate single-programmer or in-house conventions that are not widespread. Class properties based on such suffixes may be useful for that individual or organization, but they are unlikely to be valuable in a wider context. Conversely, we see that around 33% of suffixes are present in multiple program groups. We expect class properties based on these suffixes to codify community conventions or programmer instincts. Such properties should be useful in the widest context of Java programs. In our corpus, the class name suffix `List` is most widely dispersed since it is the present in 8 of the 9 program groups we examined.

3.2 Properties of Micro Patterns in the Corpus

Now we use Gil and Maman's micro pattern tool to relate class name suffixes with micro patterns. The tool is `mp-20060306`. This performs a simple static analysis on Java bytecode class files or jar archives. The tool only takes in the order of minutes to process all 29398 classes in our training corpus. For each class, it reports class metadata and a binary array of 29 values for the micro patterns which this class exhibits. Entry i is set to 1 if the class exhibits the ith micro pattern, or 0 otherwise. Figure 3 shows some sample lines of output from the micro pattern tool.

Table 3 shows that each of the 29 micro patterns is exhibited by some classes in the corpus. The relative preva-

```
org.apache.xml.serialize.HTMLdtd,dacapo-2006-10-MR2,N,19,0,0,0,0,0,0,0,1,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,1
org.apache.batik.transcoder.svg2svg.SVGTranscoder$DoctypeValue,dacapo-2006-10-MR2,N,2,0,1,0,0,0,0,0,0,0,0,1,0,0,0,0,1,0,0,0,0,0,0,0,1,0,0,0,0
net.sourceforge.pmd.rules.design.SwitchDensityRule$1,dacapo-2006-10-MR2,N,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1
org.w3c.dom.svg.SVGAnimatedPreserveAspectRatio,dacapo-2006-10-MR2,Yes,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,1,0,0
```

Figure 3. Extract of text output from Gil and Maman's micro pattern identifier tool, operating on the DaCapo benchmark jar archive file. The micro pattern array is shown as elements 6–34 inclusive on each line of comma-separated values.

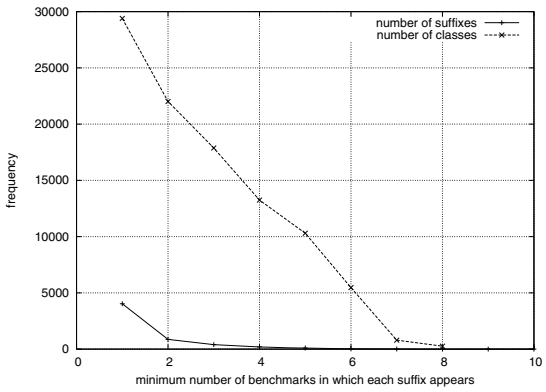

Figure 2. Graph showing how suffix groups become less significant when they must be shared across multiple program groups.

lence of each micro pattern generally agrees with the figures quoted in Gil and Maman's original research (Table 4 in [6]). In addition, we count the number of classes that do not exhibit any micro patterns. There are 12668 non-exhibiting classes in our corpus. This accounts for 43% of the total number of classes, which means that the micro-pattern coverage score for our corpus is only 57%. Ideally this coverage figure would be higher, although Gil and Maman report on some code bases that have even lower coverage scores so our corpus is not unusual in any sense.

This low coverage statistic does not reduce the significance of our suffix to micro pattern relationships, since we consider *all* classes (including non-exhibiting classes) in the studies below. Our rules would not admit that a non-exhibiting class could have a suffix that would definitely imply the class should exhibit a certain micro pattern. If a non-exhibiting class did share a common suffix with an exhibiting class, then this would *reduce the confidence level* of any rules regarding that particular suffix.

Micro pattern	*percentage*
Limited Self	19.1
Implementor	13.0
Sink	12.6
Stateless	9.0
Overrider	7.7
PureType	6.8
Box	5.0
Compound Box	4.2
Function Object	3.8
Taxonomy	3.8
Extender	3.5
Canopy	3.5
Immutable	3.4
Pool	2.3
Function Pointer	1.8
State Machine	1.7
Common State	1.6
Data Manager	1.3
Recursive	1.2
Cobol Like	1.2
Restricted	1.0
Pseudo Class	0.9
Joiner	0.8
Outline	0.8
Trait	0.7
Sampler	0.6
Augmented Type	0.5
Record	0.5
Designator	0.3

Table 3. Relative frequency of each micro pattern, in relation to the total number of Java classes in the corpus.

3.3 Correspondence between Suffixes and Micro Patterns

The key measure is the correspondence between suffix and micro patterns. Naturally, this is a non-trivial relationship. If we can show that there is some strong correspondence between certain suffixes and micro patterns, then we have a basis to support our hypothesis, namely: *Type name suffix is often an indicator of micro patterns exhibited by that class.*

Recall from Section 3.2 that 43% of classes do not exhibit any micro patterns at all. On the other hand, some classes exhibit multiple micro patterns. Gil and Maman study these details extensively in their introductory paper on micro patterns [6].

With respect to suffixes, we need to consider all classes that share a common suffix. The micro patterns exhibited by the suffix will be the *intersection* of the sets of micro patterns exhibited by each class. We actually relax this definition. If a suffix s is shared by 10 classes, but only 9 of them exhibit micro pattern p, then we say that we have 90% confidence that p corresponds with s, and so on. As we increase the confidence level, the number of correspondences drops.

Figure 4 shows the trend in number of correspondences at various confidence levels. We record the number of correspondences for suffixes that occur in at least n of the benchmark groups, for n between 1 and 4. When n is set to 1, the correspondence may be local to a single benchmark program. When n is set to 4, it seems that the correspondence is widespread, and therefore almost certainly not programmer-specific. For $n > 4$, there are only a handful of correspondences.

The message from Figure 4 is clear: even though the number of correspondences reduces as we increase the confidence and coverage thresholds, there are still some useful correspondences remaining. In the next section, we present three of these high-confidence widely dispersed correspondences and endeavour to explain them.

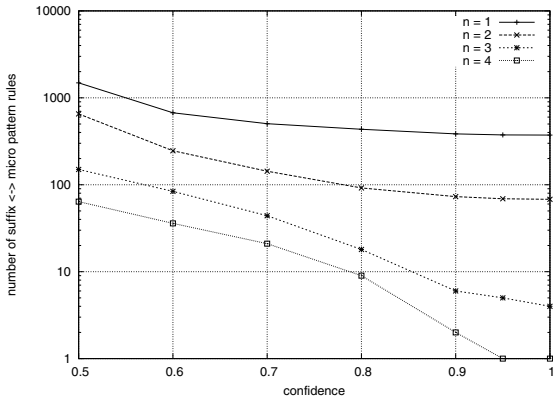

Figure 4. Graph showing how the number of micro pattern to suffix relations changes as we alter the confidence threshold. Note log scale on y-axis. These tests are for all suffix groups with two or more classes, that are seen across n benchmark groups, for four different values of n.

3.4 Benefits of Hindsight

To briefly review our achievement so far: we have analysed a corpus of Java programs and extracted correspondences between the set of class name suffixes S and the set of micro patterns MP. Given $s \in S$ and $mp \in MP$, a correspondence (s, mp) may be treated as a *rule*. The rule states that whenever a class name ends in s then that class should exhibit micro pattern mp.

This section considers three example rules that relate class name suffixes and micro patterns. These rules were generated from the corpus described above. We aim to show that the derived rules make sense. They are basically an automated identification of 'common sense' that is applied by software engineers at development time. If, given the hindsight afforded by our data mining, the rules make sense to programmers, then we can have some confidence that any programmer-help/hints system based on these rules may be useful. In addition, we suggest potential optimizations that could be enabled if these rules are observed, and potential bugs that could be occur if the rules are broken.

3.4.1 Comparable

The `Comparable` suffix is shared between 10 classes in two benchmark groups. 100% of these classes exhibit the *PureType* micro pattern. This means that the classes only contain abstract methods. They have no fields or static members. Effectively, `Comparable` classes are interfaces to be implemented.

Possible optimization: The interface membership may be encoded in a low-level manner, perhaps using spare bits in the object header. This avoids some of the runtime overhead associated with object-orientation. Note that `instanceof` tests could be optimized in the same manner.

Possible bug: If a programmer broke the convention and provided a 'default' method implementation for a Comparable class, then he may forget to override this method in a subclass, leading to incorrect behaviour of comparison operations. (Inexperienced Java programmers often make this kind of mistake with `String` equality, for instance.)

3.4.2 Assert

The `Assert` suffix is shared between nine classes in four benchmark groups. 89% of these classes exhibit the *Stateless* micro pattern. Stateless classes only have fields that are marked as `static` and `final`—i.e. constants. This is typical of assertion code, which largely consists of small methods that compare parameters against compile-time constants.

Possible optimization: Such classes are excellent candidates for aggressive constant propagation. This may not be enabled by default in an adaptive compilation environment due to the generally high cost:benefit ratio. The benefit should be greater than average for `Assert` classes.

Possible bug: A developer may break this rule by neglecting to mark fields as final in a library `Assert` class. Such fields may be updated by malicious application code to prevent the application from failing the assertions.

3.4.3 Exception

The `Exception` suffix is shared between 839 classes in six benchmark groups. 88% of these classes exhibit the *Sink* micro pattern. Methods in Sink classes do not propagate calls to any other methods. All methods are *leaf* methods.

Possible optimization: Methods in these classes are good targets for inlining. Also objects allocated in these methods are good candidates for stack allocation. Although one assumes that exception code is not normally frequently executed, it is occasionally used as a standard control flow mechanism, particularly in the lusearch benchmark from DaCapo and the _228_jack benchmark from SPECjvm98.

Possible bug: Consider a method `f` that throws exception `e`. The code to handle `e` may call a method in `e` which (perhaps unknown to the programmer) calls `f` again. This would throw the exception recursively. Again, this is a mistake a novice programmer may make, but any rules-based advice we can supply would be valuable. If we observe the rule that exceptions do not propagate calls to other methods, then such a disaster scenario is unable to occur.

4 Applications

To exploit these rules in a real-world Java development environment, we require:

1. a database that records correspondences between suffixes and micro patterns (Section 4.1)

2. a selection of tools that analyse source code and communicate information from the database to the developer (Sections 4.2 and 4.3)

4.1 Database

The database must be indexed on class name suffix. It will store a set of micro patterns associated with each suffix. In order for the user to be able to specify thresholds to eliminate less likely rules, each entry must store the number of times a particular (s, mp) correspondence holds and the number of contradictions. (This allows us to compute a percentage confidence score for each rule.)

The database must be compiled from a large training corpus of Java programs at system-install time. The developer may select these training programs, or there may be some standard set. For instance, in the case of a software company with specific in-house coding conventions, it may be useful to train on an exclusive corpus of Java programs developed by the company. In the extreme case, one could imagine a de-centralized world-wide database system, which could be queried online or cached locally.

Ideally, the database should be updated continuously as new Java programs are created. This enables the database to adapt to new programming conventions and practice, perhaps as new APIs are introduced to the Java language. In addition, it allows a developer to personalise the database by deleting rules that she does not consider helpful.

Our prototype database is stored as plain ASCII text. We search using UNIX utilities like `grep`. Obviously this approach would not scale well as database size increases.

4.2 Interactive Wizard

Our first application that uses the database is an interactive coding-time wizard, which suggests micro pattern properties for developers to consider for classes as they begin to declare those classes. As soon as the developer completes typing the class name, the interactive wizard analyses name suffix, queries the database and pops up with suggested micro patterns that this class should exhibit.

Ideally, the wizard should provide links to documentation that explains what each micro pattern entails. If the developer agrees with the wizard's suggested micro pattern(s), the wizard instruments the class source code accordingly. Perhaps the wizard might even add skeleton Java code to conform to the appropriate constraints.

Classes conforming to a micro pattern should be marked as such, either by implementing a marker interface, or by means of Java 1.5 style class-level annotations. These can be checked later, for instance by the analysis tool described in Section 4.3.

We implemented a prototype wizard in the Eclipse development environment. We extended a simple source code editor with a plugin that suggests completion proposals according to class name suffixes. If the user clicks on a completion proposal, the appropriate `implements` clause is

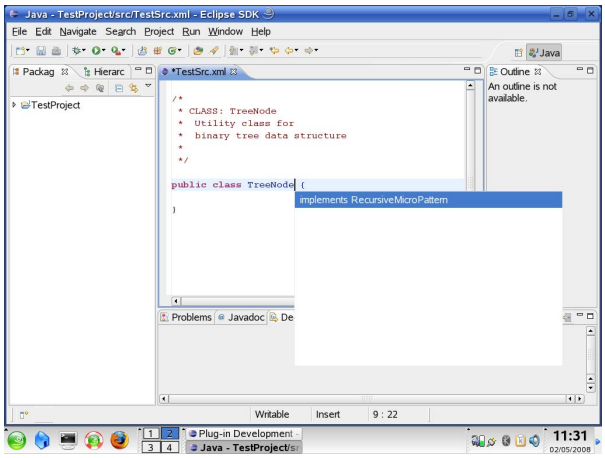

Figure 5. Screenshot of Eclipse plugin suggesting a micro pattern to be exhibited by the TreeNode class.

added to her class source code. Figure 5 shows a screenshot of the tool in action. The suffix rule states that the `Node` suffix should exhibit the *Recursive* micro pattern, which means that the class should have an instance field of the same type as the class. This is a simple linked-list or binary-tree style data structure. When there are multiple rules to apply, we can use their differing confidence levels as a relevance measure, which Eclipse uses to sort the list of suggestions presented to the user.

4.3 Lint-like Checker

The second application that uses the database is a *non-interactive checker*, akin to the lint static analysis tool. At code-review time, the checker tool is invoked by the developer, either in the development environment or on the command line. This tool outputs warnings about possible violations of suffix / micro pattern rules.

The source code may contain explicit micro pattern annotations, inserted at coding time using a tool as in Section 4.2. In this case, the tool simply checks to ensure that the specified micro patterns' constraints are not violated. If they are, an error message is reported to the user.

On the other hand, the source code may not contain explicit micro pattern annotations. In this case, the tool infers micro patterns that are implicit in the class name suffixes, based on some threshold confidence level predetermined by the user. Again, the tool checks the constraints and reports any violations back to the user.

The tool will show snippets of source code that cause micro pattern constraint violations. It may be able to provide suggested alterations to the code. This tool can be run in stand-alone mode or as part of a development environment.

Figure 6 shows the output from our tool when it discovers that a `TreeNode` class does not have a `TreeNode` field, yet it is marked as implementing the *Recursive* micro pattern.

5 Related Work

Since their introduction in 2005 [6], micro patterns have been applied throughout the fields of program analysis and software engineering.

Kim et al [8] use micro patterns to track program evolution throughout the development process. They show how to detect likely program bugs by observing changes in a class's micro patterns over time. The `Sourcerer` open source code search engine [3] employs micro patterns as a potential search criterion. Marion et al [11] use micro patterns to characterize classes at allocation sites. They construct a mapping from micro patterns to object lifetimes, which is used to optimize the allocation of long-lived objects in a generational garbage collector by means of pretenuring.

We identify correspondences between Java class name suffixes and micro patterns, which no-one else has done before. We use this information to provide support to the developer at and after coding time. Other research only uses the information after coding is complete. We assert that micro patterns can both improve runtime optimization opportunities and reduce bugs. Each of the other papers mentioned above only focuses on one of these two activities.

Høst and Østvold [7] conduct a detailed study of a massive corpus of Java applications. They focus on method naming conventions. They extract the key verb from the start of each method name, and see how this correlates with various simple method attributes. They use statistical analysis and information theory to build a lexicon of frequently occurring verbs and their attributes. This work is most similar in spirit to our own, except that they focus on method names and initial verbs, whereas we have studied class names and noun suffixes. They have produced automated documentation of existing code conventions, whereas we have used our identified conventions to implement developer-friendly tool assistants.

Our research involves predicting class properties without performing static analysis on the source code of that class. As demonstrated above, we are able to predict class properties before the source code is present. The most related work we have found is a preliminary study by Turner et al [16]. They predict low-level method properties (such as whether the method contains a for-loop) from the method signature, using a Naive Bayes classifier. They report on prediction of method properties from several packages in the standard Java API libraries. However this small pilot

```
*Violation* of the Recursive micro pattern!
Class TreeNode, declared in file:TreeNode.xml, line 9
does not contain any instance fields of type TreeNode
This rule has confidence 75%
```

Figure 6. Text output from our micro pattern checker tool working on the TreeNode source code shown earlier.

study does not seem to have been followed up, presumably due to lack of useful applications for this information.

The most active research areas for semantic information extraction are *program comprehension*, *software maintenance* and *reverse engineering*. These are large fields, so we only present a few representative examples.

Liblit et al [10] explore programming as a cognitive communication task. They analyse real-world source code identifier names. They find psychological motivation for 'meaningful' identifier names in imperative and object-oriented programming languages.

Biggerstaff et al [4] show how source code identifier names can be used to build up a set of *program concepts* that encapsulate high-level design issues in a system. Their concept assignment process is grounded in artificial intelligence techniques. Once identified, the concepts are useful for program comprehension, documentation recovery, specification reverse engineering and refactoring.

Lawrie et al [9] analyse variable identifier names in large programs written in various imperative and object oriented languages. They aim to measure consistency of variable names (which is related to the problems of synonymy and homonymy in natural language). They conclude that programming languages have a more limited use of vocabulary than natural languages.

Anquetil and Lethbridge [2] evaluate the relevance of Pascal record identifier names in a legacy telecoms application. They provide a framework to assess whether a program naming convention is reliable for use in program comprehension and reverse engineering tools.

Pollock et al [14] show how the accuracy of software searches can be improved by using natural language terms to index documents and then to construct search queries. Their main target program analysis is aspect-oriented refactoring of legacy code.

Unlike these works, we are not doing any kind of deep semantic analysis of identifiers at present. We simply associate identical class name suffixes with a common set of micro patterns. We expect that this could be used for program comprehension purposes, although that is not our primary motivation.

6 Conclusions

This paper has presented and empirically validated the hypothesis: *type name suffix is often an indicator of micro patterns exhibited by that class*. We have analysed a large corpus of real-world Java programs and extracted a set of correspondences between class name suffixes and micro patterns. This is useful for:

1. formalizing the instinctive behaviour of programmers. The rules presented in Section 3.4 make sense to Java coders, although they may not have expressed them in this way.

2. exploiting these rules for program optimization and bug detection, via development tools like our prototype Eclipse plugins described in Section 4.

For future work, we intend to repeat our analysis with other parts of Java class names apart from the suffix. Promising ideas include consideration of meaningful adjectives at the head of a name. In addition, we hope to mature our Eclipse plugins to the extent that we can use them in a *case study*, in order to determine whether such tools actually result in any quantifiable benefits in development practice or program performance.

References

[1] Eclipse project. `http://www.eclipse.org`.

[2] N. Anquetil and T. Lethbridge. Assessing the relevance of identifier names in a legacy software system. *Proceedings of the 1998 Conference of the Centre for Advanced Studies on Collaborative Research*, 1998.

[3] S. Bajracharya, T. Ngo, E. Linstead, Y. Dou, P. Rigor, P. Baldi, and C. Lopes. Sourcerer: a search engine for open source code supporting structure-based search. In *OOPSLA*, pages 681–682, 2006.

[4] T. Biggerstaff, B. Mitbander, and D. Webster. Program understanding and the concept assignment problem. *Communications of the ACM*, 37(5):72–82, 1994.

[5] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.

[6] Y. Gil and I. Maman. Micro patterns in Java code. In *OOPSLA*, pages 97–116, 2005.

[7] E. W. Høst and B. M. Østvold. The programmer's lexicon, volume I: The verbs. In *Proceedings of the Seventh IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 193–202, 2007.

[8] S. Kim, K. Pan, and E. Whitehead Jr. Micro pattern evolution. In *Proceedings of the International Workshop on Mining Software Repositories*, pages 40–46, 2006.

[9] D. Lawrie, H. Feild, and D. Binkley. Syntactic identifier conciseness and consistency. In *Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation*, pages 139–148, 2006.

[10] B. Liblit, A. Begel, and E. Sweeser. Cognitive perspectives on the role of naming in computer programs. In *Proceedings of the 18th Annual Psychology of Programming Interest Group Workshop*, 2006.

[11] S. Marion, R. Jones, and C. Ryder. Decrypting the Java gene pool: Predicting objects' lifetimes with micro-patterns. In *Proceedings of the International Symposium on Memory Management*, pages 67–78, Oct 2007.

[12] T. P. Morgan. Evans data cases programming language popularity. *The Unix Guardian*, 3(46), 2006. `http://www.itjungle.com/tug/tug121406-story03.html`.

[13] J. Noble and R. Biddle. Notes on postmodern programming. *OOPSLA Onward!*, pages 49–71, 2002.

[14] L. Pollock, K. Vijay-Shanker, D. Shepherd, E. Hill, Z. Fry, and K. Maloor. Introducing natural language program analysis. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, 2007.

[15] Sun Microsystems. Code conventions for the Java programming language, Sep 1999. `http://java.sun.com/docs/codeconv`.

[16] K. Turner, P. Cardin, S. Reid, and J. Clawson. Predicting Java source code properties from a natural language specification, 2002. Unpublished draft available from `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.91.3692`.