Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 253 (2010) 191–204
www.elsevier.com/locate/entcs

Fundamental Nano-Patterns to Characterize and Classify Java Methods

Jeremy Singer[1] Gavin Brown Mikel Luján Adam Pocock Paraskevas Yiapanis

University of Manchester, UK

Abstract

Fundamental nano-patterns are simple, static, binary properties of Java methods, such as ObjectCreator and Recursive. We present a provisional catalogue of 17 such nano-patterns. We report statistical and information theoretic metrics to show the frequency of nano-pattern occurrence in a large corpus of open-source Java projects. We proceed to give two example case studies that demonstrate potential applications for nano-patterns. The first study involves a quantitative comparison of two popular Java benchmarking suites, in terms of their relative object-orientedness and diversity. The second study involves applying machine learning techniques to program comprehension, using method nano-patterns as learning features. In both studies, nano-patterns provide concise summaries of Java methods to enable efficient and effective analysis.

Keywords: Nano-pattern, Java method

1 Introduction

Imagine you see the fragment of Java source code given in Figure 1, and you have the task of describing this method concisely to another software developer. How would you achieve this goal?

In this paper, we advocate the use of *nano-patterns* to characterise Java methods. Nano-patterns are properties of methods that are:

- *simple*: They can be detected by manual inspection from a Java developer, or by a trivial automated analysis tool.
- *static*: They should be determined by analysis of the bytecode, without any program execution context.
- *binary*: Each property is either true or false for a given method.

[1] Email: jsinger@cs.man.ac.uk

doi:10.1016/j.entcs.2010.08.042

For instance, from our current set of 17 nano-patterns, the `fib` method in Figure 1 exhibits only two nano-patterns: namely Recursive and LocalReader. Note that information is also conveyed by the fact that certain patterns are *not* exhibited: examples include ObjectCreator and Looping.

1.1 Patterns

At the high level, *design patterns* [11] encapsulate developer practice, whether that be existing conventions or managerial aspirations for better practice. These design patterns are described in terms of software architecture, using technical prose or UML diagrams. Such patterns describe re-usable templates for structuring software. Due to their high level nature, they are not directly executable or verifiable.

Recently there has been much interest in *automatic* detection of *low level patterns*, particularly in static analysis of Java bytecode. Gil and Maman [12] introduce the concept of *micro patterns* to characterize Java classes. They propose the formulation of *nano-patterns* to characterize methods within Java classes, however they do not elaborate on this idea. Høst and Østvold [15] provide a set of simple Java method attributes, which we term *fundamental nano-patterns*. In this paper, we extend Høst and Østvold's attribute set to give a fuller catalogue of fundamental nano-patterns. These patterns encapsulate Java language-specific idioms that are the *lingua franca* for experienced software developers. It must be emphasized that this catalogue is still *provisional*; we anticipate adding new nano-patterns over time.

There are many potential applications for these kinds of low level patterns. The list below mentions a number of applications that have been the subject of recent research investigations.

(i) Catalogues of idioms to enable novice developers to gain experience at reading and writing code [15,16].

(ii) Tools to detect bugs from anomalies in pattern usage and interactions [17,20].

(iii) Auto-completion hints in development environments [20].

(iv) Succinct characterization of code [3].

(v) Empirical evaluation of coding styles and standards in a common framework [12].

(vi) Relating dynamic program behaviour with patterns, to guide just-in-time op-

```
int fib(int x) {
    if (x<=1)
      return 1;
    else
      return fib(x-1) + fib(x-2);
}
```

Fig. 1. Fragment of Java source code to be characterized concisely

timization decisions [18].

1.2 Contributions

The key contributions of this paper are:

(i) A categorized catalogue of fundamental nano-patterns, each with a clear definition that would enable simple mechanical detection of the pattern from bytecode, Section 2.

(ii) Formal evaluations of the nano-pattern catalogue, using information theory (Section 3) and data mining (Section 4) techniques.

(iii) Two case studies that demonstrate how nano-patterns can be used to compare different code bases (Section 5) or to aid program comprehension via large-scale statistical analysis of Java methods (Section 6).

2 Nano-Pattern Catalogue

Nano-patterns are simple properties exhibited by Java methods. They are *traceable*; that is, 'they can be expressed as a simple formal condition on the attributes, types, name and body' of a Java method [12]. They should be automatically recognisable by a trivial static analysis of Java bytecode.

Høst and Østvold [15] present a catalogue of *traceable attributes* for Java methods. They argue that these attributes could be used as building blocks for defining nano-patterns. In this paper, we refer to these traceable attributes as *fundamental nano-patterns*, which could potentially be combined to make *composite nano-patterns*.

We have supplemented Høst and Østvold's original catalogue of fundamental nano-patterns [15]. The full set of our fundamental nano-patterns is given in Table 1. The original patterns are given in plain typeface, and our new patterns are given in bold typeface. Another novelty is that we have grouped these patterns into four intuitive categories.

It is easy to see how *composite* nano-patterns could be constructed from logical combinations of *fundamental* nano-patterns. For instance, the PureMethod nano-pattern might be specified as:

$\neg$ FieldWriter $\wedge$ $\neg$ ArrayWriter $\wedge$ $\neg$ ObjectCreator $\wedge$ $\neg$ ArrayCreator $\wedge$ Leaf

A more complex definition of method purity would remove the leaf method restriction, and replace it with the recursive constraint that all method calls must also be pure methods. However this definition would require whole-program analysis, which is considered non-trivial and therefore not suitable for a nano-pattern. Note that in the remainder of this paper, we restrict attention to *fundamental* nano-patterns only.

category	*name*	*description*
Calling	NoParams	takes no arguments
	NoReturn	returns `void`
	Recursive	calls itself recursively
	SameName	calls another method with the same name
	Leaf	does not issue any method calls
Object-Orientation	ObjectCreator	creates `new` objects
	FieldReader	reads (static or instance) field values from an object
	FieldWriter	writes values to (static or instance) field of an object
	TypeManipulator	uses type casts or instanceof operations
Control Flow	**StraightLine**	no branches in method body
	Looping	one or more control flow loops in method body
	Exceptions	may throw an unhandled exception
Data Flow	**LocalReader**	reads values of local variables on stack frame
	LocalWriter	writes values of local variables on stack frame
	ArrayCreator	creates a new array
	ArrayReader	reads values from an array
	ArrayWriter	writes values to an array

Table 1
Catalogue of fundamental nano patterns. Boldface names are for original patterns we have devised, all other patterns come from Høst and Østvold's catalogue.

2.1 Detection Tool

We have developed a command line tool to detect nano-patterns for methods in Java bytecode class files, based on the ASM bytecode analysis toolkit [6]. Our tool reads in a class file name specified as a command line argument, and dumps out a bitstring of nano-patterns exhibited for each method in the class. The detection tool is written in Java; it is only 600 source lines of code. Our code makes extensive use of data structures and visitor code from the ASM API. The tool operates in two different ways to detect specific nano-patterns:

(i) Some patterns are found by simple iteration over a method bytecode array, searching for specific bytecode instructions that indicate particular nano-patterns. For example, the `newarray` bytecode indicates the ArrayCreator nano-pattern.

(ii) Other patterns are found by simple regular expression matches on method signatures. For example, if the method type signature contains the string `()` then the method exhibits the NoParams nano-pattern.

We envisage that it should be possible to automate the generation of ASM-based detection code for specific nano-patterns, given some kind of formal specification of the nano-pattern characteristics. A meta-language like JTL [8] may be useful here. We do not address this issue in the current research.

2.2 Statistics

We analyse a large and varied corpus of Java programs; the details are given in Table 2. 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.

program	version	description
Ashes Suite	1st public release	Java compiler test programs
DaCapo	2006-10-MR2	Object-oriented benchmark suite
JBoss	3.2.2	Application server
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 nano-pattern coverage study

nano-pattern	% coverage
LocalReader	89.4
StraightLine	63.6
FieldReader	51.4
Void	50.6
NoParams	39.2
SameName	32.4
LocalWriter	31.1
ObjectCreator	26.5
FieldWriter	26.5
Leaf	20.3
TypeManipulator	15.2
Exceptions	13.6
Looping	11.3
ArrayReader	6.7
ArrayCreator	5.4
ArrayWriter	5.3
Recursive	0.7
Overall	100.0

Table 3
Coverage scores for each nano-pattern on the corpus of Java programs

In total, there are 43,880 classes and 306,531 methods in this corpus. We run our nano-pattern detection tool on all these classes. Table 3 summarises the results. It gives the proportion of methods that exhibit each kind of nano-pattern. The *overall coverage* represents the percentage of all analysed methods that exhibit any nano-pattern. Since this score is 100%, all methods analysed exhibit at least one nano-pattern from our catalogue. The mean number of nano-patterns per method is 4.9.

3 Information Theoretic Characterization

Information theoretic entropy measures the *uncertainty* associated with a random variable. In this section, we consider our nano-pattern detector tool as a black box supplying values that represent nano-pattern bitstrings. For each of the different potential bitstrings, there is an associated probability based on its frequency of

occurrence. (We estimate probabilities by frequencies in our corpus of 306,531 methods.) Given the set of all possible bitstrings B, we denote the probability of the occurrence of a particular bitstring $b \in B$ as p_b. We compute the entropy H (after Shannon) as:

$$H = -\sum_{b \in B} p_b \log_2(p_b)$$

A low entropy score indicates low uncertainty in the underlying random variable, which means that nano-patterns are very predictable. This would reduce their utility for classification. On the other hand, a high entropy score indicates high uncertainty. The maximum entropy score is $\log_2|B|$ where $|B|$ is the number of potential bitstrings. Since there are 17 different nano-patterns in our catalogue, the maximum entropy score would be 17. This would mean all nano-patterns are independent, and have a 50% chance of being exhibited by a method.

In fact, from the 306,531 methods we measured, the entropy of the bitstrings is **8.47**. This value is relatively high, which means the nano-patterns for a method are not easily predictable. There are some inter-dependencies between patterns, but these are generally non-trivial. (The next section describes cross-pattern relationships in detail.)

4 Data Mining Characterization

4.1 Background

Data mining is 'the non-trivial extraction of implicit, previously unknown, and potentially useful information from data' [10]. A number of techniques exist to perform data mining on large data sets. One of the most popular techniques is *association rule* mining from sets of items in a data set, introduced by [1]. Association rules are obtained via *frequent pattern mining*. Association rules take the form of logical implications. Their primary use is for *market basket analysis*, where vendors search for items that are often purchased together [5].

We are interested in sets of nano-patterns that are frequently exhibited together, by Java methods. Such association rules have the form $A \rightarrow B$, meaning that if method m exhibits nano-pattern A, then this implies m also exhibits B. Along with each rule, there are two related measures of interest: *support* and *confidence*. The support is the proportion of methods that exhibit both A and B in relation to the total number of methods analysed. The confidence is the proportion of methods that exhibit both A and B in relation to the total number of methods that exhibit A. A rule is only retained if it satisfies user-determined minimum thresholds for both support and confidence [2].

4.2 Nano-Pattern Analysis

We perform association rule mining on the complete set of 306,531 methods for which we have nano-pattern data. The rule mining algorithm produces hundreds of rules. However we immediately discard all rules involving the LocalReader nano-

pattern; since it is such a prevalent pattern, any rules involving it are not really meaningful. Many rules remain after this initial pruning. Some of these are obvious, for instance: ArrayCreator implies ArrayWriter with high confidence. In the remainder of this section, we report on three interesting rules that occur due to common Java programming idioms. Each of these rules exceeds our thresholds for support and confidence. We carry out further statistical analysis using the *lift* and χ^2 measures to determine whether there are statistically significant correlations between the associated nano-patterns in each rule. In each case we find that the nano-patterns are significantly positively correlated.

(1) $$\mathsf{Looping} \rightarrow \mathsf{TypeManipulator}$$

This rule is caused by the prevalence of `java.util.Iterator` objects used in while loops over data structures from the Java Collections framework. The code listing below gives an outline example.

```
while (i.hasNext()) {
  Element e = (Element)i.next();
  // ...
}
```

In older versions of Java, all objects are coerced to the `Object` supertype when they are stored in library container data structures. Even with addition of generics in Java 5 source, type casts are still present in Java bytecode for retrieving objects from container data structures. Therefore this rule is an idiomatic artifact of the Java source to bytecode transformation.

(2) $$\mathsf{ArrayReader} \rightarrow \mathsf{Looping}$$

This rule is caused by the idiom of iterating over an entire array, reading each element. The code listing below gives an outline example.

```
for (int i=0; i<a.length; i++) {
    // ...
    doWork(a[i]);
    // ...
}
```

(3) $$\mathsf{FieldWriter} \wedge \mathsf{StraightLine} \rightarrow \mathsf{NoReturn}$$

This rule is due to the prevalence of object-oriented *setter* accessor methods. Such methods take a single argument, write this value to a field of the current object and return `void`. The code listing below gives an outline example. One would expect to see this kind of rule for well-written programs in any object-oriented language.

```
public void setXYZ(Foo xyz) {
    this.xyz = xyz;
    return;
}
```

4.3 Applications

There are many potential applications for these kinds of association rules. We outline three areas below.

(i) Detection of high-level design patterns from low-level nano-patterns. In general, design pattern discovery is acknowledged to be difficult [14,9]. We have shown above that some combinations of low-level features are potential indicators for higher-level patterns. Gueheneuc et al [13] explore this concept further, although with a possibly more restrictive set of static code features.

(ii) A 'Programmer's Lexicon' style guidebook for novice programmers [15], outlining common and idiomatic programming conventions. Each discovered convention requires manual annotation to provide some measure of *goodness*. In particular, it is likely that prevalent *anti-patterns* may be discovered.

(iii) Identification of potential bugs. Given a large and varied corpus of code, we can extract a set of high-confidence association rules. If these rules are not kept in new code, an online interactive checker can inform the developer of the rule violations [20].

5 Case Study A: SPECjvm98 vs DaCapo

In this section, we use nano-patterns to contrast two Java client-side benchmark suites. In general, it is difficult to quantify the differences between two sets of programs: However we demonstrate that nano-patterns provide a good basis for differentiation.

The *SPECjvm98* benchmark suite was originally intended to evaluate the performance of commercial Java virtual machine (JVM) implementations. However due to its small size and relative age, it is now only used as a target for academic research such as *points-to analysis* [21]. A potential replacement for SPECjvm98 is the *DaCapo* benchmark suite, compiled by an academic research group. The DaCapo introductory paper [4] presents an extensive empirical study to highlight the differences between these two benchmark suites. The authors claim that DaCapo is superior to SPECjvm98 for two main reasons:

(i) DaCapo programs are more object-oriented than SPECjvm98.

(ii) DaCapo programs are more diverse in their behaviour than SPECjvm98.

Using our nano-patterns catalogue, we should be able to provide new quantitative evaluations of these criteria for the two benchmark suites.

5.1 Object Orientation

The DaCapo paper [4] argues that the DaCapo suite is 'more object-oriented' than SPECjvm98. The static analysis study that backs up this claim employs Chidamber and Kemerer metrics [7]. We can evaluate the level of static object orientation in each benchmark suite, by considering the four nano-patterns that deal with object

	benchmark	# methods	% OC	% FR	% FW	% TM	% cov
SPECjvm98	_201_compress	44	13	65	52	0	86
	_202_jess	673	33	50	23	8	75
	_205_raytrace	173	16	58	40	2	86
	_209_db	34	38	79	50	32	94
	_213_javac	5601	29	61	26	10	77
	_222_mpegaudio	280	17	60	38	2	79
	_227_mtrt	177	15	57	39	2	85
	_228_jack	302	23	36	49	10	66
	geomean	249	21	57	38	5	81
DaCapo	antlr	1788	41	62	39	13	81
	bloat	2718	33	66	33	22	85
	chart	4182	33	59	26	12	82
	eclipse	5385	27	58	29	16	79
	fop	5180	24	46	32	7	76
	hsqldb	2767	21	58	22	12	72
	jython	6549	25	55	19	19	75
	luindex	963	28	56	33	9	79
	lusearch	1252	27	58	32	10	81
	pmd	4923	20	45	26	13	66
	xalan	5512	20	54	28	10	75
	geomean	3180	27	56	28	12	77

Table 4
Object-oriented nano-pattern coverage for each benchmark

orientation. Recall from Table 1 that these are ObjectCreator, FieldReader, FieldWriter and TypeManipulator. (In this study we abbreviate these nano-patterns as OC, FR, FW and TM respectively.)

Table 4 presents the results of this analysis. For each benchmark suite, we consider every Java application separately. For each application, we perform static analysis on all methods defined in benchmark classes that are loaded by a JVM during an execution of that benchmark with the default workload. From this analysis, we report the proportion of methods that exhibit each OO nano-pattern. We also report the overall OO coverage, which gives the proportion of methods that exhibit at least one OO nano-pattern.

From these results, it is not immediately clear to see whether DaCapo is more object-oriented than SPECjvm98. They have similar overall coverage scores for the OO nano-patterns, in relative terms. However note that absolutely, DaCapo is much larger than SPECjvm98. The OO metrics given in the original DaCapo paper were absolute figures too.

A higher proportion of methods create objects in DaCapo, and it also has many more type manipulating methods. These are clear indications of object orientation. On the other hand, there are similar amount of object field reading for both suites. Interestingly, SPECjvm98 seems to perform much more object field writing. We investigate the difference between accesses to static and instance fields, since FR and FW cover both static and instance accesses by definition. Again we found similar statistics in both suites: around 20% of reads are to static fields, and less than 10% of writes are to static fields.

One potential limitation of this study is that the nano-pattern catalogue does not presently capture all object-oriented behaviour. For instance, we do not have any measure of method overriding via virtual method calls. Also we make no distinction between accessing object fields through a `this` pointer and other pointers. Perhaps a richer set of nano-patterns would provide a clearer picture.

5.2 Diversity

Nano-patterns can be used to indicate similarity between methods; we assert that similar methods should exhibit similar nano-patterns. The DaCapo paper [4] criticizes the SPECjvm98 benchmarks for being overly similar. The authors take a set of architectural metrics for each benchmark and perform a principal components analysis with four dimensions. They show that the DaCapo programs are spread around this 4-d space, whereas the SPECjvm98 programs are concentrated close together.

Again, we can use nano-patterns to confirm the results of this earlier study. We consider all nano-patterns in our catalogue from Table 1. Again, we consider all methods from benchmark classes loaded during execution. To demonstrate that different benchmarks within a suite are diverse, we take two measurements for each benchmark.

(i) *Number of unique nano-pattern bitstrings*: Given a set of nano-pattern bitstrings for a single benchmark, which of these bitstrings do not appear in any other benchmark in the suite? This characterizes behaviour that is unique to one benchmark. We can count the number of such unique bitstrings as an indicator of benchmark diversity within a suite.

(ii) *Information theoretic entropy*: Given a set of nano-pattern bitstrings for each benchmark, we can compute the information theoretic entropy of that set. High entropy values indicate greater uncertainty, i.e. the bitstrings are less predictable. Again, this can indicate benchmark diversity within a suite.

Table 5 reports the results for this analysis of benchmark diversity. It is clear to see from the geometric mean scores for each benchmark suite that DaCapo benchmarks have more unique nano-pattern bitstrings per benchmark, and that the entropy of nano-pattern bitstrings is higher for DaCapo. This analysis confirms the claims in the original DaCapo paper [4] that the DaCapo suite is more diverse than SPECjvm98.

5.3 Caveats

Analysis based on nano-patterns is entirely *static*. For a true comparison between the benchmark suites (especially in relation to diversity) it would be better to look at both static and dynamic behaviour. The DaCapo study focused entirely on dynamic behaviour, whereas we have only looked at static behaviour here. However we reach the same conclusions in relation to intra-suite diversity.

On the other hand, we assert that it is still useful to perform a static compar-

	benchmark	# methods	# unique NP sets	entropy
SPECjvm98	_201_compress	44	6	4.69
	_202_jess	673	52	6.09
	_205_raytrace	173	0	4.55
	_209_db	34	8	4.79
	_213_javac	5601	628	8.13
	_222_mpegaudio	280	32	6.48
	_227_mtrt	177	0	4.58
	_228_jack	302	24	4.92
	geomean	248.63	13.65	5.41
DaCapo	antlr	1788	28	7.22
	bloat	2718	49	7.02
	chart	4182	98	7.17
	eclipse	5385	95	8.35
	fop	5180	32	7.01
	hsqldb	2767	144	8.29
	jython	6549	136	7.13
	luindex	963	10	7.62
	lusearch	1252	13	7.65
	pmd	4923	44	7.57
	xalan	5512	110	8.08
	geomean	3179.85	50.14	7.54

Table 5
Measurements of benchmark diversity in terms of unique nano-pattern sets and nano-pattern entropy

ison of the benchmark suites in isolation. Often these particular Java benchmarks are used to compare static analysis techniques (as opposed to runtime JVM performance) in which case, static object orientation and diversity become the main concern. Hence this style of empirical comparison based on nano-patterns is indeed valuable.

6 Case Study B: Method Clustering based on Nano-Patterns

Clustering is a form of unsupervised learning. It is used to group data points into a variable number of clusters based upon a similarity measure, usually a distance metric. This enables a quick characterisation of data into higher level groupings. In this particular context, we aim to cluster similar methods to enable program comprehension, where method similarity is based on nano-pattern bitstrings. There are two main obstacles:

(i) All our nano-pattern features are binary values, which is non-standard for clustering algorithms that generally operate on real-valued continuous data.

(ii) Our nano-pattern feature space has 17 dimensions. This makes it difficult to visualize any clusterings.

To work around these problems, we use principal components analysis (PCA) to project our data into a continuous 2-d space. PCA transforms the data into a different space. It creates new features out of the axes of maximum variation in the

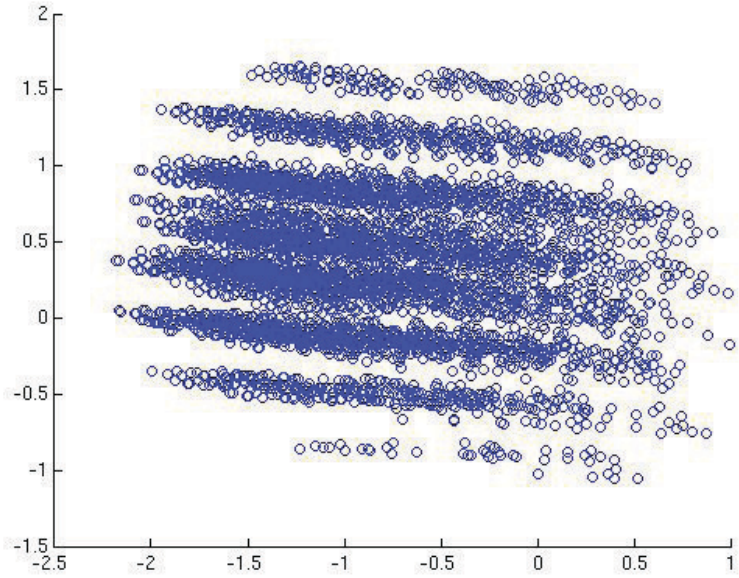

Fig. 2. 2-d projected nano-pattern data for methods in corpus (note sausage-shaped clusters)

original data set. This means the largest principal components contain the most information about the data. Figure 2 shows a visualization of this projected data. The first two principal components form the axes for this graph, as these account for most of the variation in the data.

The figure shows a number of different clusters, indicating that there are several groups of similar methods in the original data set. A further clustering on this data would provide a basis for relating the apparent clusters to the presence of combinations of nano-patterns in the original data set.

We note in passing that there has been previous work using clustering to analyse Java methods [19]. However our set of static method features appears to be richer than in earlier work. The application area for this analysis is mostly *program comprehension*.

7 Conclusions

In this paper, we have shown that fundamental nano-patterns can provide succinct characterizations of Java methods. We have demonstrated the capabilities of nano-patterns to provide a framework for quantitative analysis of large Java applications, and to enable learning-based techniques like data mining and clustering.

Our future work includes extending the provisional catalogue of nano-patterns. We hope to improve its object-oriented features with support for method overloading, overriding and `super()` calls. We also want to enrich our Exceptions nano-pattern to distinguish between methods that throw exceptions directly, catch exceptions, and propagate uncaught exceptions. Additional higher-level method

characteristics include threading activity and use of standard Java APIs like the collections framework.

Finally, we hope to employ state-of-the-art clustering algorithms to group related methods together and analyse these results. Eventually we aim to use fundamental nano-patterns in a supervised learning context.

References

[1] Agrawal, R., T. Imielinski and A. Swami, *Mining association rules between sets of items in large databases*, in: *Proceedings of the International Conference on Management of Data*, 1993, pp. 207–216.

[2] Agrawal, R. and R. Srikant, *Fast algorithms for mining association rules*, in: *Proceedings of the 20th International Conference on Very Large Databases*, 1994, pp. 487–499.

[3] Bajracharya, S., 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: *Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming, Systems, Languages, and Applications*, 2006, pp. 681–682.

[4] Blackburn, S. M. et al., *The DaCapo benchmarks: Java benchmarking development and analysis*, in: *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2006, pp. 169–190.

[5] Brin, S., R. Motwani, J. Ullman and S. Tsur, *Dynamic itemset counting and implication rules for market basket data*, in: *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, 1997, pp. 255–264.

[6] Bruneton, E., R. Lenglet and T. Coupaye, *ASM: a code manipulation tool to implement adaptable systems*, in: *Adaptable and Extensible Component Systems*, 2002.

[7] Chidamber, S. and C. Kemerer, *A metrics suite for object oriented design*, IEEE Transactions on Software Engineering **20** (1994), pp. 476–493.

[8] Cohen, T., J. Y. Gil and I. Maman, *Jtl: the Java tools language*, in: *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2006, pp. 89–108.

[9] Dong, J. and Y. Zhao, *Experiments on design pattern discovery*, in: *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, 2007, p. 12.

[10] Frawley, W., G. Piatetsky-Shapiro and C. Matheus, *Knowledge discovery in databases: An overview*, AI Magazine (1992), pp. 213–228.

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

[12] Gil, Y. and I. Maman, *Micro patterns in Java code*, in: *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2005, pp. 97–116.

[13] Gueheneuc, Y., H. Sahraoui and F. Zaidi, *Fingerprinting design patterns*, in: *Proceedings of the 11th Working Conference on Reverse Engineering*, pp. 172–181.

[14] Heuzeroth, D., T. Holl, G. Högström and W. Löwe, *Automatic design pattern detection*, in: *Proceedings of the 11th IEEE International Workshop on Program Comprehension*, 2003, pp. 94–103.

[15] Høst, E. W. 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*, 2007, pp. 193–202.

[16] Høst, E. W. and B. M. Østvold, *The Java programmer's phrase book*, in: *Proceedings of the First International Conference on Software Language Engineering*, 2008, pp. 322–341.

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

[18] Marion, S., 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*, 2007, pp. 67–78.

[19] Rousidis, D. and C. Tjortjis, *Clustering data retrieved from Java source code to support software maintenance: A case study*, in: *9th European Conference on Software Maintenance and Reengineering*, 2005, pp. 276–279.

[20] Singer, J. and C. Kirkham, *Exploiting the correspondence between micro patterns and class names*, in: *Proceedings of the Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*, 2008, pp. 67–76.

[21] Sridharan, M. and R. Bodík, *Refinement-based context-sensitive points-to analysis for Java*, in: *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2006, pp. 387–400.