Micro Patterns in Java Code[*]

Joseph (Yossi) Gil[†] Itay Maman
Department of Computer Science
Technion—Israel Institute of Technology
{ yogi | imaman } @ cs.technion.ac.il

Abstract

Micro patterns are similar to design patterns*, except that micro patterns stand at a lower, closer to the implementation, level of abstraction. Micro patterns are also unique in that they are mechanically recognizable, since each such pattern can be expressed as a formal condition on the structure of a class.*

This paper presents a catalog of 27 micro-patterns defined on JAVA *classes and interfaces. The catalog captures a wide spectrum of common programming practices, including a particular and (intentionally restricted) use of inheritance, immutability, data management and wrapping, restricted creation, and emulation of procedural-, modular-, and even functional- programming paradigms with object oriented constructs. Together, the patterns present a set of prototypes after which a large portion of all* JAVA *classes and interfaces are modeled. We provide empirical indication that this portion is as high as 75% .*

A statistical analysis of occurrences of micro patterns in a large software corpus, spanning some 70,000 JAVA *classes drawn from a rich set of application domains, shows, with high confidence level that the use of these patterns is not random. These results indicate consciousness and discernible design decisions, which are sustained in the software evolution. With high confidence level, we can also show that the use of these patterns is tied to the specification, or the purpose, that the software realizes.*

The traceability*,* abundance *and the* statistical significance *of micro pattern occurrence raise the hope of using the classification of software into these patterns for a more founded appreciation of its design and code quality.*

Categories and Subject Descriptors

D.3 [**Software**]: Programming Languages

General Terms

Design, Object-Oriented Programming

[*]Research supported in part by Israel Science Foundation (ISF) grant no. 2004460.
[†]Work is supported in part by the IBM faculty award.

Keywords

Program Analysis, Design Patterns, Implementation Patterns

1. Introduction

We all know what makes one algorithm better than another: time, space, random-bits, disk access, etc. are established, *objective* and well defined metrics [14] to be employed in making such a judgement. In contrast, the assessment of quality of software design is an illusive prospect. Despite the array of books and research articles on the topic (see e.g., [11, 12, 28, 30]), a question such as "*Is Design A better than Design B?*" can, still, only be decided by force of the argumentation, and ultimately, by the personal and *subjective* perspective of the judge.

The research described in this paper is concerned with the important, yet so recalcitrant, problem of finding sound objective methods of assessment of design. Medical experiments can prove that a certain medication is better than another in treating a specific ailment. We all want to carry similar controlled experiments to prove that certain design methods are more likely to produce better software than others. However, in contrast with many other natural sciences, experiments on large scale software development are so prohibitively costly that much of the research on the topic abandoned this hope.

Our attack on this multiple Gordian knot is by taking a different angle at it: Rather than subjecting the development process to experimentation, we apply statistical tools to *existing* artifacts of the development. Instead of dealing with "is *A better than B?*" sort of questions, our research should help in rigorously determining *"how is A different than B?*'. We can also show that certain design techniques are more common than others. The judgement of the quality of design can perhaps then be reduced to the judgement of the abundance of the design, and the quality of the software that uses it.

This angle is made possible by the bountiful class structure of JAVA [3], together with the colossal, publicly available, base of software in the language, which opens the road for sound claims and understanding of the way people write software (more precisely, on the software written by people). We argue that this class structure makes it possible to find traces of design, specifically of what we shall call *micro patterns*.

1.1 Traceability of Design

Can design be traced and identified in software? The prime candidates of units of design to look for in the software are obviously *design patterns* [22]. However, despite the dozen years that passed since the original publication [21], and the voluminous research ensuing it, attempts to automate and formalize design patterns are scarce. Systems like DisCo [31], LePUS [16,17], SPINE and HEDGEHOG [6], constraint diagrams [27], Elemental Design Patterns [39], and others did not gain much popularity. Specific research on detection of design patterns exhibited low precision, typically with high rate of false negatives (see e.g., [9, 25]). Indeed, as Mak, Choy and Lun [29] say, "*. . . automation support to the utilization of design patterns is still very limited*".

Traceable Patterns. To overcome this predicament, we define the notion of *traceable patterns*, which are similar to design patterns, except that they are mechanically recognizable and stand at a lower level of abstraction. A pattern is traceable if it can be expressed as a *simple formal condition on the attributes, types, name and body of a software module and its components*.

It is required that these patterns are not random; they must capture a non-trivial *idiom* of the programming language which serves a *concrete purpose*. Yet, by definition, traceable patterns stand at a lower level of abstraction than that of the classical collection of design patterns [22]. This is because traceable patterns are tied to the implementation language and impose a condition on a single software module.

Micro Patterns. Traceable patterns can be defined on many kinds of modules, including code fragments, routines, classes and packages. We coin the term *micro patterns* as a short hand for "*class-level traceable patterns*". This paper is concerned with micro patterns, and specifically in the context of the JAVA programming language. When no confusion can arise we shall, for the sake of brevity, call these just patterns.

We present a catalog of 27 micro patterns, organized in 8 categories, including idioms for a particular and intentionally restricted use of inheritance, immutability, wrapping and data management classes, object oriented (OO) emulation of procedural, modular and even functional programming paradigms, innovative use of class structure, and many more.

Examples. A simple example for concreteness is the Sampler pattern in the *Controlled Creation* category. This pattern defines classes which have a **`public`** constructor, but in addition have one or more **`static public`** fields of the same type as the class itself. The purpose of such classes is to give clients access to pre-made instances of the class, but also to create their own. The Sampler is realized by, e.g., class `Color` from package `java.awt` of the JAVA standard runtime environment, which offers a spectrum of pre-defined colors as part of its interface.

Another example of a micro pattern is the Immutable pattern [24] in the *Degenerate State* category. This pattern prescribes an object whose state cannot be changed after its construction.

The reader is invited to take a sneak preview at the entire list of patterns in Sec. 3 for further examples.

1.2 Micro patterns and Productivity

Other than serving for a more rigorous study of design, our catalog, just as many other collection of patterns, can help in *documentation*, in conveying a *knowledge base*, and in setting a *vocabulary* for communication among and between coders and designers.

The vocabulary that the catalog sets can come handy in the description of implementation strategies of design patterns. Terms such as Immutable, Box, Canopy, Pure Type or Implementor (all patterns from the catalog) are useful in describing the implementation of design patterns such as DECORATOR, BRIDGE, PROXY, etc. On a broader perspective, software frameworks may use this terminology to describe the various sorts of classes which take part in the framework.

Our empirical study demonstrates the consistent abundance of each of the patterns; further, the entire catalog characterizes about three quarters of the classes in our corpus. These finding support the claim that micro patterns can enhance the following aspects of software engineering productivity:

- *More Efficient Design.* The catalog captures a substantial body of knowledge gathered from a massive software corpus. The use of this knowledge base can make the design and implementation stages more efficient, by using one of the recipes in the catalog, rather than designing a class from scratch.

  The mental effort saved by using familiar, named patterns for certain classes, can be redirected to more important and difficult tasks.

- *Code Learning and Reuse.* Familiarity with the catalog makes it possible for programmers to quickly understand an overwhelming majority of the JAVA software base. They can then focus more attention to the smaller fraction of the remaining code, which presumably requires closer examination.

- *Training.* By learning the patterns in the catalog, programmers can be quickly introduced to the tools of trade of JAVA programming.

- *Automation.* Micro-patterns traceability makes it also possible to *enrich* automatically generated documentation produced by tools such as JavaDoc[1].

1.3 New Language Constructs

The restrictive nature of the patterns in the catalog, combined with their abundance might tell that the myriad of combinations by which the different features of the underlying programming language is too great.

Consider the many different kinds of fields that a JAVA class can have: they can be **`static`** or non-**`static`**, **`final`** or not-**`final`**, inherited or introduced by the class, and they can exhibit one of four different kinds of visibility. Methods show an even greater variety, since they can also be **`abstract`**, **`final`**, overriding, or even *refining*; and, there are also constructor methods, and anonymous static initializers, . . . Our count shows that there are over forty different kinds of class members, without even considering variety due to type signature or naming.

Micro patterns are not only patterns of class design. They are also patterns (in the information theoretical sense of the term) by which the programmer makes selections from this huge space of different combinations of class features. By recognizing that the expressive power of the programming language might be too large, we may be lead not only to a more structured system of teaching design, but also of maturing some of these combinations into full blown language constructs.

The precise definition of micro patterns makes it possible to evolve some of the patterns into language constructs, in the manner suggested by Agerbo and Cornils [1] regarding the incorporation of design patterns into a programming language (interestingly, the motivation for JAVA's new `enum` facility is reflected by the prevalence of Augmented Type and Pool micro pattern).

1.4 Statistical Inference

We next pose several research questions regarding the use of micro patterns in practice. In the paper we apply statistical inference in an attempt to answer these. For this purpose, we assembled a large corpus of JAVA software from 14 different software collections, from four different application domains. In total, the corpus spanned over seventy thousand classes. The statistical analysis was

[1]`http://java.sun.com/j2se/javadoc`

carried out on the results of matching the micro patterns against the corpus.

Langauge Constraints or Software design? We found that micro patterns are abundant in JAVA. A basic question is whether the existence and choice of use of these patterns is an artifact of langauge design rather than software design? In other words, we would like to know whether this abundance reflects true design decisions rather than following the invisible tracks and paths that the JAVA designers have set in the language.

The analysis of our empirical results suggest that the latter is very unlikely. Specifically, the findings in Sec. 8 indicate that with high confidence level ($\alpha < 0.01$), it is not the case that patterns occur in JAVA with a fixed random probability, irrespective of the programming context.

Specification Design or Software Design? Another related question is whether the choice of patterns is decreed by the software specification, rather than being a matter of choice of the implementor. The answer we give in Sec. 10 is that both factors contribute to this choice.

On the one hand, at high confidence level (in the statistical sense of this phrase), we can claim that different implementations of the same specification will *tend* to use the same patterns.

On the other hand, the vendor or software team has a degree of freedom in making this choice, i.e., not all implementations of the same specification will follow the same combination of patterns.

Moreover, we can claim, and again with the soundness of statistical high confidence, that the exercise of this degree of freedom is not random, but rather an expression of a specific and discernible style.

Sec. 10 also shows that the choice of pattern for each class is sustained in different editions of the same software.

Individual Value of Each of the Patterns? A fundamental question regarding micro patterns is whether each of the patterns in the catalog is valuable. For example, one may argue that patterns such as Pseudo Class and Cobol like capture bad coding practices and thus should not be included in our catalog. A similar reasoning may suggest that the abundance of patterns such as Designator or Record is so small that they deliver no practical value to a programmer.

It turns out that our research can provide an objective, quantitative answer to this question of *"the value of a single pattern"*: We show that each of the patterns matters for the statistical distinction between software products of different origins.

Furthermore, we show that since a class can be characterized by more than one pattern, the catalog is, in a sense, greater than the sum of its parts.

Specifically, we define an information theoretical metric of the amount of design information, i.e., the number of bits, that the catalog reveals on a software collection. Our experiments indicate that in average the catalog provides about 5 bits of design information on the classes in our corpus, and, this number is greater than the sum of the information that the individual patterns provide (Sec. 7).

Can Patterns be Used to Characterize Software? As explained above, the question of whether the existence of patterns matter to the quality of the design or the coding is in a sense philosophical. The reason is that we do not yet have sufficient objective means of quality evaluation.

Still, even when these means mature, or if we suffice ourselves with a subjective evaluation, an important question is whether the *choice of patterns makes a significant and meaningful influence on the software.*

Our answer to this question is two fold.

1. We show that the catalog touches a great deal of the software: Three out of four classes can be characterized by the catalog; many carry even more than one pattern label (see Sec. 7).

2. We show that the differences between pattern prevalence levels in different software collections are significant (Sec. 8).

Therefore, we have that the patterns as a whole are significant in characterizing a large portion of software collections. Of course, we cannot show that the characterization of software by patterns, i.e., these 5 bits of information, are directly tied with quality. Nonetheless, one may still be able to draw conclusions from the fact that a pattern is used extensively in software that came from respected vendors such as Sun or the Apache group.

The question of the extent by which the micro patterns in the catalog *contributes* to the global (or local) software quality, is difficult, and must be left open for debate or further research, in which the finding of this paper, as well as the statistical methods we employ may become useful.

Outline. The remainder of this paper can be divided to three parts. The first part is concerned with the description of micro patterns: Sec. 2 gives an overview of the patterns catalog, while the individual patterns are described in detail in Sec. 3. Sec. 4 then elaborates further on the notion, including a comparison to design patterns, and to what may be called *implementation patterns* [4].

The second part is dedicated to the experiments. Sec. 5 defines the notion of entropy of a pattern. The software corpus is described in Sec. 6, while the core of the experimental results are in Sec. 7.

In the third part of this paper, we move on to statistical analysis of the empirical results. Sec. 8 employs statistical tests to check whether the difference in prevalence levels are significant. Sec. 9 is an intermediate discussion which helps in understanding the significance of the differences. Sec. 10 continues the statistical investigation, by checking whether the patterns make statistically significant distinction between different versions of the same software.

Related work is the subject of Sec. 11. Sec. 12 reflects back on the results, and raises directions for further research.

2. The Micro Patterns Catalog

To compare micro patterns to design patterns, implementation patterns, and other kinds of patterns, we need to become familiar with the micro patterns themselves.

This section overviews the micro patterns *catalog*, briefly presenting the patterns. A more detailed description of the patterns is in the subsequent Sec. 3.

2.1 The Construction of the Catalog

The process by which the catalog was conceived may be instructive for understanding its structure. Variations on this process may help in finding other micro patterns which we have missed.

The search for micro patterns started by considering the various kinds of features that a JAVA class may have. We tried then to work out meaningful and useful restrictions of the freedom in using these features. To do so, we raised questions such as "*how could a class with no fields be useful?*", "*are there any classes of this sort in existence?*", "*how can these classes be characterized?*". Conversely, having thought of a useful programming practice, we tried to translate it into a condition on the code, and then inspect classes that matched this condition.

We implemented each of these initial "pre-patterns" and applied them to the classes in the corpus. Manual inspection of the code of these classes lead to a refinement of some of the definitions, abandonment of others, merges and splits of others, until the catalog reached its current shape.

Thus, the search for patterns started from definitions, which lead to code inspection, and then to the refinement of the definitions.

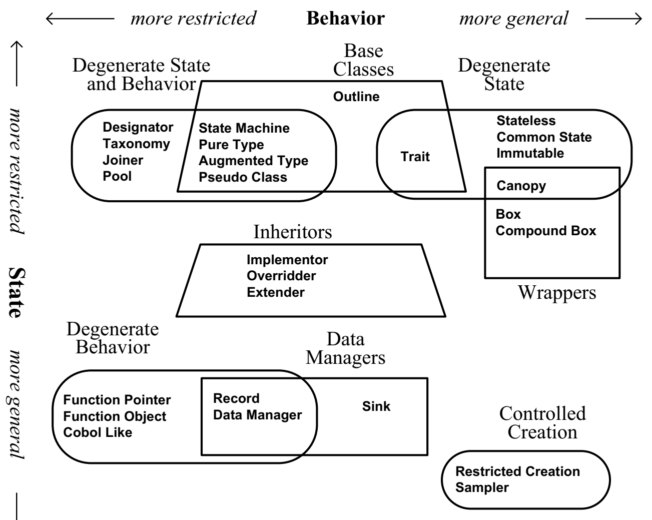
It is tempting to do the converse, i.e., cluster the existing code base, and discover patterns in it, devoid of any a priori dictations. To do so we, we tried several approaches. For example, we broke down the conditions we already discovered into atomic predicates ("basic features" in the learning lingo), such as "number of instance fields is 1", "no superinterfaces", etc.

The values of these predicates on the classes in the software corpus were then fed into an associations rules analyzer [47]. In return, the analyzer generated long lists of dependencies between these predicates, sorted in descending order of strength.

Unfortunately, none of these dependencies revealed something that we could have interpreted as a purposeful pattern. Other established techniques of machine learning did not work for us.

2.2 The Structure of the Catalog

Consider Fig. 1, which shows a global map of the catalog, including the 8 categories, and the placement of the 27 micro patterns into these. (The patterns themselves are described in brief in Tab. 1.)



Rounded rectangles denote pattern categories in which state, behavior, or construction is degenerate, rectangles denote categories of patterns for containment, while trapezoids denote patterns used for inheritance.

Figure 1: A map of the micro patterns catalog

The X-dimension of Fig. 1 corresponds to class behavior. Categories at the left hand side of the map are those of patterns which restrict the class behavior more than patterns which belong to categories at the right.

Similarly, the Y-dimension of the figure corresponds to class state: Categories at the upper portion of the map are of patterns restricting the class state more than patterns which belong to categories at the bottom of the map.

Altogether, there are four categories (depicted as rounded corners rectangles in the figure) in which the class behavioral, or creational or variability (state) aspects of a class are degenerate: *Degenerate State and Behavior*, *Degenerate State*, *Degenerate Behavior* and *Restricted Creation*.

Depicted as rectangles in Fig. 1, there are two categories pertaining to containment: The *Data Managers* category is that of patterns which directly store and manage data; The *Wrappers* category contains patterns which wrap other classes.

Finally, there are also two categories pertaining to inheritance: *Base Classes* and *Inheritors*. These categories are portrayed as trapezoids in the figure.

Tab. 1 gives an alternative, textual representation of the information depicted in Fig. 1. As can be seen in the table (and also in Fig. 1), the categories are not disjoint. There are a number of patterns which belong in two categories. The last column of the table shows the additional category of such patterns.

For example, Pseudo Class pattern belongs both to the *Degenerate State and Behavior* and the *Base Classes* categories; Pure Type is a *Base Class* which also exhibits *Degenerate State and Behavior*. Such patterns are described in one of their categories, and merely mentioned in the others.

This table also tersely describes each of the patterns. It is important however to note that this one line description, by nature, cannot be precise or complete. To see that, recall that there are many, not necessarily disjoint, kinds of methods which JAVA admits, including inherited methods, static methods, concrete methods, abstract methods, constructors, etc.

There are therefore several ways of translating a simple statement such as *"all methods are public"* into a precise and complete condition on the code. For example, one needs to decide whether the universal quantification in this statement precludes inheriting protected methods.

Hence, the descriptions presented in Tab. 1 should serve merely as an intuitive summary. The definitions in the forthcoming Sec. 3 provide a more precise description of the micro patterns.

3. Description of the Micro Patterns

In this section, we will try to explain better the *purpose* of each pattern, give an example of its use, and derive a more precise definition out of it. Still, for space limitations, we cannot provide the full formal expression of the condition of each pattern.

The largest patterns bulk is described in Sec. 3.1, which is concerned with the *Degenerate State and Behavior*, *Degenerate State*, *Degenerate Behavior* and *Restricted Creation* categories.

Containment-based patterns are described in Sec. 3.2. Patterns related to inheritance, i.e., *Base Classes* and *Inheritors* are the subject of Sec. 3.3.

3.1 Degenerate Classes

Out of the 27 patterns in the catalog, there are 21 patterns in which the state, behavior or the creation are degenerate in one way or another. This section describes the 12 patterns out of these which have no other category. The remaining 9 patterns are described together with their other respective category.

3.1.1 Degenerate State and Behavior

The first, and most simple category of micro patterns, includes those interfaces and classes in which both state and behavior are extremely degenerate. This degeneracy means, in most cases, that the class (or interface) does not define any variables or methods. Despite these severe restrictions, classes and interfaces which fall into this group are useful in tasks such as making and managing global definitions, class tagging, and more generally for defining and managing a taxonomy.

In addition to the patterns listed below, this category also contains the Pure Type, Augmented Type, Pseudo Class, and State Machine micro patterns which are described in the *Base Classes* category.

1. Designator. The most trivial interface is an empty one. Interestingly, vacuous interfaces are employed in a powerful programming technique, of tagging classes in such a way that these tags can be examined at runtime.

For example, a class that **`implements`** the empty interface `Clone-`

	Main Category	Pattern	Short description	Additional Category
Degenerate Classes	Degenerate State and Behavior	Designator	An interface with absolutely no members.	
		Taxonomy	An empty interface extending another interface.	
		Joiner	An empty interface joining two or more superinterfaces.	
		Pool	A class which declares only static final fields, but no methods.	
	Degenerate Behavior	Function Pointer	A class with a single public instance method, but with no fields.	
		Function Object	A class with a single public instance method, and at least one instance field.	
		Cobol Like	A class with a single static method, but no instance members	
	Degenerate State	Stateless	A class with no fields, other than static final ones.	
		Common State	A class in which all fields are static.	
		Immutable	A class with several instance fields, which are assigned exactly once, during instance construction.	
	Controlled Creation	Restricted Creation	A class with no public constructors, and at least one static field of the same type as the class	
		Sampler	A class with one or more public constructors, and at least one static field of the same type as the class	
Containment	Wrappers	Box	A class which has exactly one, mutable, instance field.	
		Compound Box	A class with exactly one non primitive instance field.	
		Canopy	A class with exactly one instance field that it assigned exactly once, during instance construction.	Degenerate State
	Data Managers	Record	A class in which all fields are public, no declared methods.	Degenerate Behavior
		Data Manager	A class where all methods are either setters or getters.	
		Sink	A class whose methods do not propagate calls to any other class.	
Inheritance	Base Classes	Outline	A class where at least two methods invoke an abstract method on "this"	Degenerate State
		Trait	An abstract class which has no state.	
		State Machine	An interface whose methods accept no parameters.	Degenerate State and Behavior
		Pure Type	A 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	A class which can be rewritten as an interface: no concrete methods, only static fields	
	Inheritors	Implementor	A concrete class, where all the methods override inherited abstract methods.	
		Overrider	A class in which all methods override inherited, non-abstract methods.	
		Extender	A class which extends the inherited protocol, without overriding any methods.	

Table 1: Micro patterns in the catalog

`able`[2] indicates (at run time) that it is legal to make a field-for-field copy of instances of that class.

Thus, a Designator micro pattern is an interface which does not *declare* any methods, does not *define* any static fields or methods, and does not inherit such members from any of its superinterfaces.

A class can also be Designator if its definition, as well as the definitions of all of its ancestors (other than `Object`), are empty.

Pattern Designator is the rarest, with only 0.2% prevalence in our software corpus. It was included in the catalog because it presents an important JAVA technique, which is also easily discernible.

2. Taxonomy. Even if the definition of an interface is empty it may still extend another, potentially non-empty, interface.

Consider for example interface `DocAttribute` (defined in package `javax.print.attribute`). This interface extends interface `Attribute` in the same package without adding any further declarations. Interface `DocAttribute` is used, similarly to the Designator micro pattern, for tagging purposes—specifically that the attribute at hand is specialized for what is known as "Doc" in the JRE.

An empty interface which extends a single interface is called a Taxonomy, since it is included, in the subtyping sense, in its parent, but otherwise identical to it.

There are also classes which are Taxonomy. Such a class must similarly be *empty*, i.e., add no fields nor methods to its parent. Since constructors are not inherited, an empty class may contain constructors. A Taxonomy class may not implement any interfaces.

This micro pattern is very common in the hierarchy of JAVA's exception classes, such as: `EOFException` which extends `IOException`. The reason is that selection of a **`catch`** clause is determined by the runtime type of the thrown exception, and not by its state.

3. Joiner. An empty interface which extends more than one interface is called a Joiner, since in effect, it joins together the sets of members of its parents.

For example, the interface `MouseInputListener` joins together two other interfaces: interface `MouseMotionListener` and interface `MouseListener`.

An empty class which implements one or more interfaces is also a Joiner. For example, class `LinkedHashSet` marries together class `HashSet` and three interfaces `Cloneable`, `Serializable` and `Set`.

4. Pool. The most degenerate classes are those which have neither state nor behavior. Such a class is distinguished by the requirement that it declares no instance fields. Moreover, all of its declared static fields must be final[3]. Another requirement is that the class has no methods (other than those inherited from `Object`, or automatically generated constructors).

[2]This, and all subsequent examples are drawn from the standard Java Runtime Library.

[3]If a class has **`final`** instance fields, then, each of its instances may have a different (immutable) state, and therefore it cannot be characterized as having no state.

A Pool is a class defined by these requirements. It serves a the purpose of grouping together a set of named constants.

Programmers often use **`interface`**s for the Pool micro pattern. For example, package `javax.swing` includes interface `SwingConstants` which defines constants used in positioning and orienting screen components.

The pattern, also called *"constant interface anti-pattern"* [7], makes it possible to incorporate a name space of definitions into a class by adding an **`implements`** clause to that class.

3.1.2 Degenerate Behavior

The degenerate behavior category relates to classes with no methods at all, classes that have a single method, or classes whose methods are very simple.

5. Function Pointer. Very peculiar are those classes which have no fields at all, and only a single **`public`** instance method.

An example is class `LdapNameParser` (which is defined in package `com.sun.jndi.ldap.LdapNameParser`). This class has a single `parse` method, with (as expected) a string parameter.

Instances of Function Pointer classes represent the equivalent of a function pointer (or a pointer to procedure) in the procedural programming paradigm, or of a function value in the functional programming paradigm. Such an instance can then be used to make an indirect polymorphic call to this function. The task of function composition (as in the functional programming paradigm), can be achieved by using two such instances.

6. Function Object. The Function Object micro pattern is similar to the Function Pointer micro pattern. The only difference is that Function Object has instance fields (which are often set by the class constructor). Thus, an instance of Function Object class can store parameters to the main method of the class.

The Function Object pattern matches many anonymous classes in the JRE which implement an interface with a single method. These are mostly event handlers, passed as callback hooks in GUI libraries (AWT and Swing). Hence, such classes often realize the COMMAND design pattern.

7. Cobol like. Formally, the Cobol like micro pattern is defined by the requirement that a class has a single static method, one or more static variables, and no instance methods or fields. This particular programming style makes a significant deviation from the object oriented paradigm. Although the prevalence of this pattern is vanishingly small, instances can be found even in mature libraries.

Beginner programmers may tend to use Cobol like for their main class, i.e., the class with function

```
public static void main(String[] args)
```

The prevalence of Cobol like is not high, standing at the 0.5% level in our corpus. However, we found that it occurs very frequently (13.1%) in the sample programs included with the JAVA Tutorial [10] guides.

The *Degenerate Behavior* category also includes two other patterns: Record, which has no methods at all, and Data Manager, in which all methods are either setters or getters. The two also belong in the *Data Managers* category, and are described below (Sec. 3.2.2) with the other patterns of that category.

3.1.3 Degenerate State

The *Degenerate State* category pertains to classes whose instances have no state at all, or that their state is shared by other classes, or that they are immutable. In this category we also find the Trait pattern which is defined under its other, *Base Classes*, category (Sec. 3.3.1), and the Canopy pattern (defined under *Wrappers*).

8. Stateless. If a class has no fields at all (except for fields which are both **`static`** and **`final`**), then it is stateless. The behavior of such a class cannot depend on its history. Therefore, the execution of each of its methods can only be dictated by the parameters.

Micro pattern Stateless thus captures classes which are a named collection of procedures, and is a representation, in the object-oriented world, of a software library in the procedural programming paradigm.

A famous example of the Stateless micro pattern is the `Arrays` class, from package `java.util`.

9. Common State. At the next level of complexity, stand classes that maintain state, but this state is shared by all of their instances. Specifically, a class that has no instance fields, but at least one static field is a Common State.

For example, the class `System` manages (among other things) the global input, output, and error streams.

A Common State with no instance methods is in fact an incarnation of the *modular*[4] *programming paradigm* in the JAVA world.

10. Immutable. An immutable class is class whose instance fields are only changed by its constructors.

The Canopy is an immutable class which has exactly one instance field. Its description is placed under its other category, *Wrappers* (Sec. 3.2.1). More general is the Immutable micro pattern, which stands for immutable classes which have *at least two* instance fields.

Class `java.util.jar.Manifest` is an Immutable class since assignment to its two fields takes place only in constructors code.

3.1.4 Controlled Creation

There are two patterns in this category, which match classes in which there is a special protocol for creating objects.

The first pattern prevents clients from creating instances directly. The second pattern provides to clients ready made instances.

11. Restricted Creation. A class with no public constructors, and at least one **`static`** field of the same type as the class, matches the Restricted Creation micro pattern.

Many SINGLETON classes satisfy this criteria. A famous example is `java.lang.Runtime`.

12. Sampler. The Sampler matches classes class with at least one public constructor, and at least one **`static`** field whose type is the same as that of the class. These classes allow client code to create new instances, but they also provide several predefined instances.

An example is class `Color` (in package `java.awt`) with fields such as `red`, `green` and `blue`.

3.2 Containment

We identified six patterns by which classes manage their internal fields. There are three patterns in the *Wrappers* category, concerned with classes in which there is a principal field. The case of multiple fields is covered by the three patterns in the *Data Managers* category.

[4]The term "modular" means here module-oriented, where modules are software units such as ADA [44] packages and modules in MODULA [37].

3.2.1 Wrappers

Wrappers are classes which wrap a central instance field with their methods. They tend to delegate functionality to this field. The main pattern in this category is Box. The case that the wrapper protects the field from changes is covered by Canopy. There are cases in which there is an auxiliary field; these are captured by pattern Compound Box.

13. Box. A Box is class with exactly one instance field. This instance field is mutated by at least one of the methods, or one of the static methods, of the class.

Class `CRC32` (in the `java.util.crc` package) is an example of this micro pattern. Its entire state is represented by a single field (`int crc`), which is mutated by method

```
update(int i)
```

14. Canopy. A Canopy is a class with exactly one instance field which can only changed by the constructors of this cass.

The name Canopy draws from the visual association of a transparent enclosure set over a precious object; an enclosure which makes it possible to see, but not touch, the protected item.

Class `Integer`, which boxes an immutable **`int`** field, is a famous example of Canopy.

As explained above, since the Canopy pattern captures immutable classes, it also belongs in the *Degenerate State* category.

15. Compound Box. This is a variant of a Box class with exactly one non-primitive instance field, and, additionally, one or more primitive instance fields. The highly popular `Vector` class matches the Compound Box pattern.

3.2.2 Data Managers

Data managers are classes whose main purpose is to manage the data stored in a set of instance variables.

16. Record. JAVA makes it possible to define classes which look and feel much like PASCAL [46] record types. A class matches the Record micro pattern if all of its fields are **`public`** and if has no methods other than constructors and methods inherited from `Object`.

Perhaps surprisingly, there is a considerable number of examples of this pattern in the JAVA standard library. For example, in package `java.sql` we find class `DriverPropertyInfo` which is a record managing a textual property passed to a JDBC driver.

17. Data Manager. Experienced object-oriented programmers will encapsulate all fields of a Record and use setter and getter methods to access these.

We say that a class is a Data Manager if all of its methods (including inherited ones) are either setters or getters[5].

Recall that Data Manager micro pattern (just as the previously described Record) also belong to the *Degenerate Behavior* category.

18. Sink. A class where its declared methods do not call neither instance methods nor static methods is a Sink.

Class `JarEntry` of package `java.util.jar.JarEntry` is an example of Sink.

3.3 Inheritance

Finally, we have eight micro patterns which capture some of the common techniques by which classes prepare for inheritance (the *Base Classes* category), or interact with their superclass (the *Inheritors* category).

[5]We used the most conservative approach for the detection of such methods.

3.3.1 Base Classes

This category includes five micro patterns capturing different ways in which a base class can make preparations for its subclasses.

19. Outline. An Outline is an abstract class where two or more declared methods invoke at least one abstract methods of the current ("this") object.

For example, the methods of `java.io.Reader` rely on the abstract method

```
read(char ac[], int i, int j)
```

Obviously, Outline is related to the TEMPLATE METHOD design pattern.

20. Trait. The Trait pattern captures abstract classes which have no state. Specifically, a Trait class must have no instance fields, and at least one abstract method.

The term Trait follows the traits modules of Schärli, Ducasse, Nierstrasz and Black [38]. A trait module, found in e.g., the SCALA [35] programming language, is a collection of implemented methods, but with no underlying state.

For instance, class `Number` (of package `java.lang`) provides an implementation for two methods: `shortValue()` and for method `byteValue()`. Other than this implementation, class Number expects its subclass to provide the full state and complement the implementation as necessary.

21. State Machine. It is not uncommon for an interface to define only parameterless methods. Such an interface allows client code to either query the state of the object, or, request the object to change its state in some predefined manner. Since no parameters are passed, the way the object changes is determined entirely by the object's dynamic type.

This sort of interface, captured by the State Machine pattern, is typical for state machine classes.

For example, the interface `java.util.Iterator` describes the protocol of the standard JAVA iterator, which is actually a state machine that has two possible transitions: `next()` and `remove()`. The third method, `hasNext()` is a query that tests whether the iteration is complete. In the state machine analogy, this query is equivalent for checking if the machine's final state was reached.

22. Pure Type. A class that has absolutely no implementation details is a Pure Type. Specifically, the requirements are that the class is abstract, has no static members, at least one method, all of its methods are abstract, and that it has no instance fields. In particular, any interface which has at least one method, but no static definitions is a Pure Type.

An example is class `BufferStrategy`, which is found in package `java.awt.image.BufferStrategy`. As the documentation of this class states, it "*represents the mechanism with which to organize complex memory ...*". The concrete implementation can only be fixed in a subclass, since, "*Hardware and software limitations determine whether and how a particular buffer strategy can be implemented.*". Indeed, this class has nothing more than four abstract methods which concrete subclasses must override.

23. Augmented Type. There are many interfaces and classes which declare a type, but the definition of this type is not complete without an auxiliary definition of an *enumeration*. An enumeration is a means for making a new type by restricting the (usually infinite) set of values of an existing type to smaller list whose members are individually enumerated.

Typically, the restricted set is of size at least three (a set of cardinality two is in many cases best represented as **`boolean`**).

For example, methods `execute` and `getMoreResults` in interface `java.sql.Statement` take an **`int`** parameter that sets their mode of operation. Obviously, this parameter cannot assume any integral value, since the set of distinct behaviors of these methods must be limited and small. This is the reason that this interface gives symbolic names to the permissible values of this parameter.

Formally, an Augmented Type is a Pure Type except that it makes three or more **`static final`** definitions of the same type.

Pattern Augmented Type pattern is quite rare (0.5%), probably thanks to the advent of the Enum mechanism to the language.

24. Pseudo Class. A Pseudo Class is an abstract class, with no instance fields, and such that all of its instance methods are **`abstract`**; **`static`** data members and methods are permitted. A Pseudo Class could be *mechanically* rewritten as an interface. For instance, class `Dictionary`, the abstract parent of any class which maps keys to values, could be rewritten as an interface.

Pseudo Class is an "anti-pattern" and is not so common; its prevalence is only 0.4%.

3.3.2 *Inheritors*

The three disjoint patterns in this category correspond to three different ways in which a class can use the definitions in its superclass: implementing abstract methods, overriding existing methods and enriching the inherited interface. The catalog does not include patterns for classes which mix two or more of these three.

25. Implementor. An Implementor is a non-abstract class such that all of its the public methods were declared as abstract in its superclass.

An example is class `SimpleFormatter`, which is defined in the `java.util.logging` package). This class has single public method,

```
format(LogRecord logrecord),
```

which was declared abstract by the superclass, `Formatter` (of the same package).

26. Overrider. A class where each of its declared public methods overrides a non-abstract method inherited from its superclass. Such a class changes the behavior of its superclass while retaining its protocol. A typical Overrider class is the `BufferedOutputStream` class.

27. Extender. An Extender is a class which extends the interface inherited from its superclass and super interfaces, but does not override any method.

For example, class `Properties` (in `java.util`) extends its superclass (`Hashtable`) by declaring several concrete methods, which enrich the functionality provided to the client. None of these methods overrides a previously implemented method, thus keeping the superclass behavior intact. Note that an Extender may be regarded as an instantiation of a degenerate mixin class [8] over its superclass.

4. On the Nature of Micro Patterns

Having described the patterns themselves, we are now ready to discuss the *notion* of micro patterns in some more detail.

As explained above, micro patterns are a kind of traceable patterns.

DEFINITION 1. *A traceable pattern is a* condition on the attributes, types, name and body of a module and its components, *which is recognizable (mechanically), purposeful, prevalent and simple.*

Micro patterns are special in that the condition described in the condition Def. 1 applies to classes and interfaces. Lying outside the scope of this paper are patterns whose condition applies to other kinds of modules. We propose the term *nano-patterns* for traceable patterns which stand at the method or procedure level. The term *milli-patterns* can then be used for traceable patterns at the package level (or to any other kind of class grouping or mode of cooperation).

Sec. 4.1 explains the four characteristics of traceable patterns. Next, Sec. 4.2, discuss the differences between micro patterns and design patterns, which are mostly due to the difference in abstraction level. Finally, Sec. 4.3 discusses the notion of *implementation patterns*, and compares these with micro-patterns.

4.1 Traceable Patterns

We next discuss in greater detail the four properties of traceable patterns: *recognizability*, *purposefulness*, *prevalence* and *simplicity*.

Recognizability. The term *"mechanically recognizable"* means that there exists a Turing machine which decides whether any given module matches this condition. Thus, a condition "the module delegates its responsibilities to others" is not recognizable. On the other hand a predicate such as "each method invokes a method of another class with the same name", can be automatically checked.

Purposefulness. By *purposeful* we mean that the condition defining a micro pattern characterizes modules which fulfill a recurring need in a specific manner. The condition that the number of methods is divisible by the number of fields does not constitute a pattern. In contrast, micro pattern Canopy describes classes which have a single instance field that is assigned only once, at construction time. This idiom typically serves the purpose of managing a single resource by a dedicated object, a practice which Stroustrup [43] calls "Resource acquisition is initialization".

Prevalence. The *prevalence* of a pattern (with respect to a certain collection of modules) is the portion of modules matched by this pattern. Prevalence is an important indication that a micro pattern is purposeful[6].

The lower (and upper) bound on prevalence is determined by common sense. The programming technique captured by the Designator pattern is so unique that a prevalence of 0.2% is acceptable. The prevalence of Implementor, on the other hand, is circa 30%. Since patterns are made for distinguishing unique properties, we will tend to negate the pattern definition if its prevalence is greater than 50%.

Simplicity. The *simplicity* requirement is not only a matter of aesthetics: By sticking to first order predicate logic (FOPL), whenever possible restricted in Horn clauses form, should make it easier for the pattern recognizer to suggest corrections in case the pattern is violated.

Together, these four properties make traceable patterns useful: as patterns they bring value to the manual work of the software engineer in capturing a common and meaningful idiom of the programming language. Traceability, expressed in the simplicity and recognizability properties, help in automating some of the engineer's work.

4.2 Micro Patterns vs. Design Patterns

One of the difficult tasks in software development is bridging the gap which separates the initial *imprecise and informal* system re-

[6]Clearly, it is not sufficient—classes in which the number of methods is prime are prevalent, but have no common purpose.

quirement from the *precise and formal* manifestation of software in code written in a specific programming language. But even the smaller steps along the bridge over this gap cannot be all formal, precise or automatic. Design patterns make one important such step, while implementation patterns, which are formally defined construct, stand between the code and design patterns.

In most cases, a micro pattern is not a strategy of implementation of a design pattern. For example, we discover that the Compound Box micro pattern which is quite popular, is not acknowledged as a design pattern.

There are however several obvious relations between design patterns and micro patterns. For example, the Function Object micro pattern, is very useful for implementing the COMMAND design pattern; Sampler is one implementation of the FLYWEIGHT design pattern; most of the classes which realize the SINGLETON pattern will match the Restricted Creation micro pattern, etc.

Just like design pattern, implementation patterns follow an *extensional mode of definition* and satisfy the *locality criterion* (in the sense of the work of Eden and Kazman [18]). Still, there are several consequences to the fact that micro patterns stand at a lower level of abstraction:

- *Scope.* First, micro patterns are of a single software module in a particular programming language. Examining the list of micro patterns in Sec. 3 we can see that they are all about individual JAVA **class**es and **interface**s. Design patterns on the other hand are not so tied to a specific language, and often pertain to two or more classes, sometimes to an entire architecture.

- *Recognizability.* Second, a crucial property of micro patterns is that they are easily recognizable by software, which renders a smooth path to automation. Florijn, Meirjer and Winsen [20] enumerate three key issues in automating design patterns: *application*, *validation* and *discovery*. As van Emde Boas argues [45], the expressiveness of the language used for defining patterns, affects the complexity of these issues, and in particular detection.

  In using a formal language, which is at a lower level than the free text description of the semantics of design patterns, automation issues become much easier. Therefore, micro patterns are, by definition, automatically recognizable.

  We can also envisage a CASE tool which would help in their application, by offering a boilerplate to be filled by the implementor. For this reason, we try, whenever possible, to present the condition in the form of Horn clause constraints. As demonstrated (in another context) by Demsky and Rinard [15], this particular form can be used to deduce, *automatically* or (for better performance) semi-automatically, specific rules for correcting the input so that it matches a formal constraint. Such rules can be used by the CASE tool to generate useful warnings and advice to the programmer.

  The requirement that micro patterns are written in FOPL makes it possible to deterministically *check* a proof that one implementation pattern is mutually exclusive, contained, or overlapping with another.[7] In contrast, distinct design patterns, presented as solutions to two different problems, may be structured similarly, but be different in their intent. (A famous example is made by the STRATEGY and ADAPTER design patterns.) It follows that there is an inherent ambiguity in the process of discovering design patterns in software.

- *Context Existence.* Third is the observation that micro patterns do not usually provide "a solution to a problem in a context". The design problem and the context in which it occurs are not present when an implementation is carried out. Indeed, much of the work on the automation of design forgets the problem and the context.

  Micro patterns are not different. For example, the Sink micro pattern, occurring in about a sixth of all classes, is too general to be tied to a specific design problem. Nevertheless, there is value in adhering to the pattern. This practice will reducing code complexity, and promoting uniformity, decomposability, and clarity.

  Another example is the Box micro pattern, which represents a useful programming technique. Incidently, this technique occurs in many and not very related design patterns. The Box is therefore a term which can be used to describe and help understand many classes. Yet, it may serve a multitude of unrelated problems.

  A third example is the Function Pointer pattern, whose single **public** instance method may serve many different purposes. Yet, it is not easy to propose a unifying characteristic of these.

  Using the semiotic approach [34] to the interpretation of patterns, we have that in traceable patterns there is distinction between signifier and signified. A micro pattern is thus "a solution in search of a problem". It serves a concrete purpose, but the programmer is still required to find the right question.

- *Usability of Isolated Patterns.* A fourth difference, resulting from the loss of the problem and context in micro patterns, is the utility of individual patterns. Knowledge of the problem and context makes it possible for a design pattern to provide much more information on the proposed solution. Thus, even a single design pattern is useful on its own. In contrast, micro patterns are not as specific; their power stems from their organization in a catalog, a box of tools, each with its own specific purpose and utility.

  Given an implementation task, the programmer can choose an appropriate pattern from the catalog. Our empirical findings show that, in the majority of cases, such a micro pattern will be found. Admittedly, the nature of micro patterns is such that they do not provide as much guidance as design patterns. On the other hand, the guidance that a micro pattern does provide is suited for automatization, and does not rely as much on abilities of the individual taking that guidance.

- *Empirical Evidence.* Fifth, and perhaps most important is the fact that implementation patterns carry massive empirical evidence of their prevalence, their correlation with programming practices, and the amount of information they carry. With the absence of automatic detection tools, claims of the prevalence of design patterns is necessarily limited to the yield of a manual harvest.

[7]If we stick to full blown FOPL, it is impossible to automatically generate such a proof; it is possible to do so if patterns are constrained to use an appropriately selected subset of FOPL.

4.3 Micro Patterns vs. Implementation Patterns

Beck [4] presents an extensive discussion of implementation patterns. His book enumerates as many as 92 such patterns, all presented in the context of the SMALLTALK [23] programming language. These patterns touch software units of different levels: starting at patterns of message send, going through patterns for temporary variables, followed by patterns detailing method implementation, climbing up to instance variables, and ending with single class design.

Beck enumerates several roles that *implementation patterns* serve, including help in reading the code, accelerating the implementation, aid in communication between programmers and documentation.

These roles are not foreign to those of design patterns. Capturing existing lore, and means of communication are essential characteristics of all kinds of patterns.

Yet, implementation patterns come handy at a different stage of the development process. Design patterns are mostly useful at the drawing board. Implementation patterns are most effective when the programmer opens the langauge specific integrated development environment.

However, the fact that implementation patterns show up at a later stage of the development process does not mean that they are always traceable. Consider for example Beck's **Composed Method** implementation pattern. This pattern instructs the SMALLTALK programmer (indeed, a programmer in any language) to continue breaking methods into smaller parts until each method satisfies the (informal) condition that of serving a single identifiable task, and all operation in it stand at the same level of abstraction. It is difficult to fathom a simple formal predicate on the body of a method that will check whether this condition is true.

Another example is implementation pattern **Pluggable Selector** (similar to C's function pointers) which may not be easy to detect.

At the other end stand patterns such as **Query Method**, **Comparing Method**, and **Setting Method**, which are traceable. In our terminology, these are called nano-patterns.

The other important difference distinguishing micro patterns from implementation patterns is that micro patterns can be used at the late design stage as well as during the implementation. While doing class design, micro patterns can be employed to explain the kind of operations expected in inheritance, and for better characterization of the classes. At the implementation stage, the micro pattern(s) prescribed to the class can be used as a guiding recipe, which can even be checked automatically.

5. Definitions

The fact that micro patterns are defined on specific locations in the code (classes, or more generally any other module), rather than on the entire software fabric lets us make precise notions describing pattern interaction.

We denote the prevalence of a pattern p by $\xi(p)$. Let p_1 and p_2 be patterns. We say that p_1 is *contained* in p_2 if $p_1 \rightarrow p_2$; they are *mutually exclusive* if $p_1 \rightarrow \neg p_2$, i.e., a module can never match more than one of them. The *co-prevalence* of the patterns (with respect to a software collection) is the prevalence of $p_1 \wedge p_2$; they are *independent* if their co-prevalence is a product of their respective prevalence levels, i.e., $\xi(p_1 \wedge p_2) = \xi(p_1)\xi(p_2)$.

Let $P = \{p_1, \ldots, p_n\}$ be a patterns catalog. Then, the *coverage* of the catalog is the $\xi(p_1 \vee \cdots \vee p_n)$, i.e., prevalence of the disjunction of all patterns in the catalog.

A catalog is more meaningful if the patterns in it are not mutually exclusive. If this is the case, each module can be described by several patterns in the catalog, and the whole catalog can present more information than the simple classification of modules into $n+1$ categories.

We will now make precise the of amount of information that a catalog carries. First recall the definition of the information theoretical entropy.

DEFINITION 2. *Let $\xi_1, \ldots, \xi_k$ be a distribution, i.e., for all $i = 1, \ldots, k$ it holds that $0 \leq \xi_i \leq 1$, and $\sum_{1 \leq i \leq k} \xi_i = 1$. Then, the* entropy *of $\xi_1, \ldots, \xi_k$ is*

$$H = H(\xi_1, \ldots, \xi_k) = -\sum_{1 \leq i \leq k} \xi_i \log_2 \xi_i, \qquad (1)$$

where the summand $\xi_i \log_2 \xi_i$ is taken to be 0 if $\xi_i = 0, 1$.

The entropy is maximized when the distribution is to equal parts, i.e., $p_i = \frac{1}{k}$ for all $i = 1, \ldots, k$, in which case $H = \log_2 k$.

To gain a bit of intuition into Def. 2, let us apply it to a single pattern with prevalence ξ (with respect to some software collection). We can say that the pattern occurs with probability ξ, and not occur at probability $1 - \xi$, giving rise to the following entropy

$$-\xi \log_2 \xi - (1-\xi) \log_2 (1-\xi).$$

Suppose that $\xi = \frac{1}{2^n}$. Then, the first summand states that the fact that pattern does occur carries n bits of information (the event occurs in only $\frac{1}{2^n}$ of all cases), but these bits have to be weighted with the "probability" of the event. The second summand corresponds to the complement event, i.e., that the pattern does not occur.

Fig. 2 shows the entropy of a single pattern as a function of the prevalence.

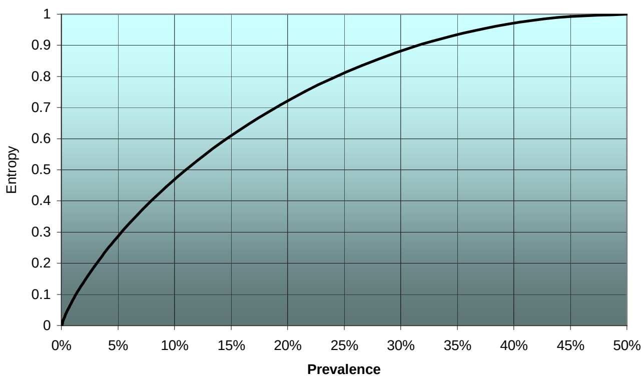


Figure 2: Entropy vs. prevalence level of a single pattern.

As the figure shows, the entropy achieves its maximal value of 1 when the prevalence is 50% and drops to zero when the prevalence is zero. The entropy is 0.72 if $\xi(p) = 20\%$, drops to 0.47 when $\xi(p) = 10\%$, to 0.29 when $\xi(p) = 5\%$, to 0.08 when $\xi(p) = 1\%$, and to 0.01 when $\xi(p) = 0.1\%$.

The entropy of an entire catalog is defined as the entropy of the distribution of the many different combinations of patterns in the catalog

DEFINITION 3. *The* entropy of a catalog P *(with respect to a certain software collection) is*

$$H(P) = -\sum_{Q \in \wp P} \xi(Q) \log_2 \xi(Q),$$

where $\wp P$ is the power set of P and $\xi(Q)$ is the prevalence of the event that all patterns in Q occur and all the patterns in $P \setminus Q$ do not *occur, i.e.,*

$$\xi(Q) = \xi\left(\bigwedge_{p\in Q} p \quad \textit{and} \quad \bigwedge_{p\in P\setminus Q} \neg p\right).$$

An entropy of (say) 4 of a catalog with respect to a certain software collection can be understood as equivalent to the amount of information obtained by a partitioning of the collection to $16 = 2^4$ equal parts. We can think of $2^{H(P)}$ as the *separation power of the catalog*.

Information theory tells us that entropy is an *additive* property in the sense that the entropy of a catalog of independent patterns is the *sum* of the entropies of each of these events. If patterns in a catalog are mutually exclusive, then the entropy of the catalog is *less* than the sum of the individual entropies.

As mentioned before, the patterns in our catalog are not mutually exclusive, which makes the catalog more informative. On the other hand, we do not expect software patterns to be truly independent. In order to evaluate the contribution of each pattern to the expressive power of the catalog, we can examine its marginal contribution to the entropy of the catalog.

DEFINITION 4. *The* marginal entropy *of pattern $p \in P$ with respect to a catalog P, written $H(p/P)$ is*

$$H(P) - H(P \setminus \{p\}).$$

The sum of the marginal entropies can be greater, smaller or equal to the entropy of the whole catalog.

If a pattern is identical to one of the other patterns in the catalog, or to any combination of these, then its marginal entropy is 0. Conversely, suppose that a certain pattern partitions every combination of the other patterns in the catalog into two equal parts. Then, the marginal entropy of this pattern is 1.

6. Data set

In the experiments, we measured the prevalence level of each of the patterns in the catalog in large collections of JAVA classes, available in the `.class` binary format. As explained in Sec. 2 the analysis was carried out by invoking a set of predicates over all classes in the collection.

A corpus of fourteen large collections of JAVA classes, totalling over three thousand packages, seventy thousand classes and half a million methods served as data set for our experiments. Tab. 2 summarizes some of the essential size parameters of these collections. The table does not include a line count of the collections in the corpus, since many of the collections are available in binary format only.

As can be seen in the table, the collections, although all large, vary in size. The smallest collection (JEdit) has about 800 classes and 6,000 methods, while the largest (JBoss) has almost a thousand packages, 18,699 classes and 157,460 methods. The median number of classes is about 4,000.

These collections can be partitioned into several groups

1. *Implementations of the standard* JAVA *runtime environment.* The JAVA runtime environment (JRE) is the language standard library, as implemented by the language vendor, which provides to the JAVA programmer essential runtime services such as text manipulation, input and output, reflection, data structure management, etc.

   We included several different implementations of the JRE in our corpus for two purposes. First, to examine the stability of patterns in the course of evolution of a library, we used the vanilla Sun implementations of versions 1.1, 1.2, 1.3, 1.4.1, and 1.4.2 of the J2SE specification. These are denoted respectively by Sun$^{1.1}$, Sun$^{1.2}$, Sun$^{1.3}$, Sun$^{1.4.1}$ and Sun$^{1.4.2}$ in Tab. 2.

   Second, to compare the incidence of micro patterns across different implementations of the same specification, we used implementations of several other vendors. The first of which is the JRE implementation included in the Kaffe project[8]–which is a non-commercial JVM implementation. Our corpus includes two versions of this implementation: Kaffe$^{1.1}$ and Kaffe$^{1.1.4}$ distinguished by their JRE version.

   We had also tried to expand our corpus with three commercial JRE libraries supplied with theses JVM products: (i) *IBM 32-bit Runtime Environment for* JAVA *2*, version 1.4.2; (ii) *J2SE for HP integrity*, version 1.4.2; and (iii) *Weblogic JRockit 1.4.2* by BEA. Eventually, these three collections were *not* included in the corpus since they all exhibited an overwhelming similarity with Sun$^{1.4.2}$. Our experiments indicated that these three were in many ways a port of the Sun implementation. Obviously, no significant data can be drawn from the analysis of these.

2. *GUI Applications.* The corpus includes two GUI applications: JEdit—which is version 4.2 of the programmer's text editor written in JAVA with a Swing GUI, and Poseidon–a popular UML modeling tool delivered by Gentleware[9]. (We used version 2.5.1 of the community edition of the product.)

3. *Server Applications.* There were two collections in this category: JBoss—the largest collection in our corpus is version 3.2.6 of the famous JBoss[10] application server (JBOSS AS) which is an open source implementation of the J2EE standard, Tomcat—part of the *Apache Jakarta Project*[11], which is a servlet container used by http servers to allow JAVA code to create dynamic web pages (version 5.0.28).

Collection	Domain	Packages	Classes	Methods
Kaffe$^{1.1}$	JRE impl.	75	1,220	10,945
Kaffe$^{1.1.4}$	JRE impl.	152	2,511	22,022
Sun$^{1.1}$	JRE impl.	67	991	9,448
Sun$^{1.2}$	JRE impl.	131	4,336	36,661
Sun$^{1.3}$	JRE impl.	170	5,213	44,747
Sun$^{1.4.1}$	JRE impl.	314	8,216	73,834
Sun$^{1.4.2}$	JRE impl.	330	8,740	76,675
Scala	Lang. tools	96	3,382	32,008
MJC	Lang. tools	41	1,141	10,927
Ant	Lang. tools	120	1,970	17,902
JEdit	GUI	23	805	6,110
Tomcat	Server	280	4,335	43,868
Poseidon	GUI	594	10,052	77,988
JBoss	Server	998	18,699	157,460
Total		3,391	71,611	620,595

Table 2: The JAVA class collections comprising the corpus.

[8]`http://www.kaffe.org`
[9]`http://www.gentleware.com`
[10]`http://www.jboss.org`
[11]`http://jakarta.apache.org`

4. *Compilers and Langauge Tools*. This category includes Ant—another component of the Apache project[12]–a build tool which offers functionality that is similar, in principle, to the popular make utility (version 1.6.2), and Scala—version 1.3.0.4 of the implementation of the Scala multi-paradigm programming language [35]; and, MJC—version 1.3 of the compiler of multiJAVA, a language extension which adds open classes and symmetric multiple dispatch to the language.

Thus, the corpus represents a variety of software origins (academia, open source communities and several independent commercial companies), interaction mode (GUI, command line, servers, and libraries), and application domains (databases, languages, text processing).

Collection	Packages	Classes	Methods
Sun$^{1.4.2}$	272	7,525	66,676
Scala	68	2,678	25,186
MJC	32	945	8,607
Ant	45	421	3,883
JEdit	21	676	4,653
Tomcat	132	1,434	14,367
Poseidon	477	8,162	61,645
JBoss	750	13,623	110,820
Shared	346	5,979	55,431
Total	2,143	41,443	351,268

Table 3: The JAVA class collections in the pruned corpus.

Note that the totals in the last line of Tab. 2 include multiple and probably not entirely independent implementations of the same classes. For experiments and calculations which required independence of the implementation, we used only collection Sun$^{1.4.2}$ out of the nine different JRE implementations, Also, as many as 5,979 classes recurred in several collections since software manufacturers tend to package external libraries in their binary distribution.

To assure independence, all such classes were pruned out of their respective collections and included in a pseudo-collection named Shared. (Interestingly, the 100 or so classes comprising the famous Junit [5] library, were found in several collections in our corpus, thus turning Shared into a super set of the Junit library.) This process defined a **Pruned** software corpus by

$$\textbf{Pruned} = \{\mathsf{Sun}^{1.4.2}, \mathsf{Scala}, \mathsf{MJC}, \mathsf{Ant}, \mathsf{JEdit}, \mathsf{Tomcat}, \mathsf{Poseidon}, \mathsf{JBoss}, \mathsf{Shared}\}.$$

The total size of this corpus and each of the (pruned) collections in it is reported in Tab. 3. We can see that the elimination of duplicates and dependent implementations halved the size of the corpus. In total, more than 41,000 independent class comprise the pruned corpus.

7. Experimental Results

The experimental results of running the pattern analyzer on the pruned corpus are summarized in Tab. 4.

The table shows the prevalence, coverage, entropy and marginal entropy of the patterns in the corpus. The body of the table presents, for each micro pattern and each software collection, the *prevalence* of the micro pattern in the collection, that is, the percentage of classes in this collection which match this micro pattern. The two last rows give a summary of each collection. (Note that due to overlap between the patterns, columns do not add up to the total coverage in the last row.) The seven last columns give summarizing statistics on each of the patterns.

In this section we take mostly a broad perspective in the inspection of this information, and will be interested in the more global properties of the catalog, including coverage, entropy, and marginal entropy. In the next section, we will march on to a deeper *statistical analysis* of this information.

Coverage. The most important information that this table brings is in the penultimate line, which shows the coverage of our catalog. We see that 79.5% of all classes in Sun$^{1.4.2}$ are cataloged. The collection with least coverage is Ant, but even for it, one in two classes is cataloged. The total coverage of the (pruned) corpus is 74.9%. The fluctuation in coverage level is not very great—the standard deviation (penultimate column) is 11%.

CONCLUSION 7.1. *Three out of four classes match at least one micro pattern in the catalog.*

The above, just as all subsequent conclusions, refer to what can be observed in our corpus. There is still a need for appropriate statistical tools to support an extrapolation of such statements to e.g., the universe of all JAVA programs.

Prevalence. In examining Tab. 4 in greater detail, we see that the most prevalent group is this of the inheritors micro patterns. About 35% of all classes not only inherit from a parent, they also adhere to a specific, particularly restrictive style of inheritance. The most common micro pattern, in this category and overall, is Implementor which occurs in about 21% of all classes. This finding indicates wide spread use of the technique of separating type and implementation, by placing the implementation in a concrete class.

Also large is Overrider, which occurs in about 11% of all classes.

A large group is also that of classes with degenerate state, whose total prevalence is about 24%.

CONCLUSION 7.2. *One in four classes is degenerate in respect to the data it maintains.*

In this group, the largest pattern is Stateless (8.9% prevalence), which is unique in that it has no instance fields.

The base class category is also quite significant, occupying about 15% of all classes. The largest pattern there is Pure Type with 10.6% prevalence.

It is interesting to see that the Sink, a class which essentially does not communicate with any other class, is also very frequent, with prevalence of 13.9%.

Together, the five leading patterns (Implementor, Sink, Overrider, Pure Type and Stateless) describe 23,848 classes, which are 58% of the classes in our pruned corpus.

CONCLUSION 7.3. *The majority of classes are cataloged by one of the five leading patterns.*

Separation Power. Conc. 7.3 does not mean that we can make do with only five patterns. The other patterns in the catalog contribute to the information it provides. One of the reasons is that the micro patterns are not mutually exclusive. There are classes in the corpus which match more than one micro pattern. Fig. 3 depicts the number of classes in the pruned corpus for each multiplicity level.

We see that 31% of the classes matched a single pattern, 30% matched two patterns, 13% matched three patterns. Out of the total 41,443 classes in this corpus there was also a significant number of

[12]`http://ant.apache.org`

Collection	Sun$^{1.4.2}$	Scala	MJC	Ant	JEdit	Tomcat	Poseidon	JBoss	Shared	Total	Average	Median	Min	Max	σ	$H(p/P)$
Designator	0.2%	0.1%	0.2%	0.0%	0.0%	0.2%	0.1%	0.3%	0.3%	0.2%	0.2%	0.2%	0.0%	0.3%	0.1%	0.05
Taxonomy	4.4%	2.7%	3.2%	1.4%	1.2%	2.6%	3.8%	3.2%	3.5%	3.5%	2.9%	3.2%	1.2%	4.4%	1.1%	0.13
Joiner	0.7%	1.8%	0.0%	0.0%	0.0%	0.6%	0.3%	2.2%	0.9%	1.2%	0.7%	0.6%	0.0%	2.2%	0.8%	0.09
Pool	1.9%	1.0%	4.6%	1.7%	1.0%	1.5%	1.7%	2.9%	2.7%	2.3%	2.1%	1.7%	1.0%	4.6%	1.1%	0.15
Sink	20.6%	14.0%	10.7%	14.3%	9.0%	12.1%	11.3%	12.7%	13.5%	13.9%	13.1%	12.7%	9.0%	20.6%	3.3%	0.67
Record	0.4%	0.3%	0.2%	0.2%	0.6%	0.3%	0.4%	1.1%	1.5%	0.8%	0.6%	0.4%	0.2%	1.5%	0.5%	0.08
Data Manager	1.8%	0.2%	1.2%	4.0%	1.5%	1.7%	1.9%	1.8%	2.4%	1.8%	1.8%	1.8%	0.2%	4.0%	1.0%	0.04
Function Pointer	2.0%	0.9%	1.8%	1.2%	1.2%	2.8%	1.7%	1.7%	1.0%	1.6%	1.6%	1.7%	0.9%	2.8%	0.6%	0.11
Function Object	7.7%	0.8%	9.1%	1.4%	24.1%	2.4%	6.3%	4.2%	5.2%	5.5%	6.8%	5.2%	0.8%	24.1%	7.1%	0.23
Cobol Like	0.4%	0.6%	0.5%	0.7%	0.1%	1.0%	0.5%	0.7%	0.4%	0.5%	0.5%	0.5%	0.1%	1.0%	0.2%	0.07
Stateless	9.8%	14.6%	7.6%	5.7%	6.1%	10.3%	6.8%	9.6%	6.8%	8.9%	8.6%	7.6%	5.7%	14.6%	2.8%	0.38
Common State	2.4%	0.3%	2.1%	0.2%	3.4%	1.3%	1.8%	7.1%	3.6%	3.8%	2.5%	2.1%	0.2%	7.1%	2.1%	0.14
Canopy	9.8%	3.9%	11.0%	4.5%	26.5%	4.6%	10.3%	6.3%	4.5%	7.7%	9.0%	6.3%	3.9%	26.5%	7.1%	0.28
Immutable	7.6%	5.6%	7.0%	2.1%	12.0%	4.0%	6.2%	6.1%	4.6%	6.1%	6.1%	6.1%	2.1%	12.0%	2.7%	0.28
Box	4.6%	14.5%	3.3%	3.1%	1.3%	8.6%	2.5%	7.8%	5.1%	6.0%	5.6%	4.6%	1.3%	14.5%	4.1%	0.22
Compound Box	6.0%	5.1%	3.6%	10.0%	5.8%	3.1%	3.8%	3.7%	4.4%	4.4%	5.0%	4.4%	3.1%	10.0%	2.1%	0.24
Implementor	26.0%	10.5%	17.8%	17.1%	37.1%	12.7%	22.1%	23.1%	15.8%	21.3%	20.2%	17.8%	10.5%	37.1%	8.1%	0.63
Overrider	12.4%	4.1%	8.1%	4.0%	23.1%	20.2%	16.8%	7.0%	9.4%	10.8%	11.7%	9.4%	4.0%	23.1%	6.9%	0.23
Extender	4.3%	1.6%	5.3%	4.8%	4.9%	5.9%	4.5%	4.2%	4.2%	4.2%	4.4%	4.5%	1.6%	5.9%	1.2%	0.23
Outline	1.8%	0.2%	1.1%	1.0%	0.4%	0.3%	1.3%	0.6%	0.6%	0.9%	0.8%	0.6%	0.2%	1.8%	0.5%	0.09
Trait	1.3%	0.3%	0.8%	0.2%	0.0%	0.7%	0.8%	0.4%	0.6%	0.7%	0.6%	0.6%	0.0%	1.3%	0.4%	0.08
State Machine	1.5%	1.8%	1.0%	0.7%	0.3%	1.7%	1.7%	2.1%	1.8%	1.8%	1.4%	1.7%	0.3%	2.1%	0.6%	0.09
Pure Type	7.7%	20.5%	6.7%	3.1%	2.5%	5.6%	11.9%	11.2%	10.1%	10.6%	8.8%	7.7%	2.5%	20.5%	5.5%	0.15
Augmented Type	0.6%	0.0%	0.3%	0.5%	0.0%	0.1%	0.2%	0.4%	1.0%	0.5%	0.4%	0.3%	0.0%	1.0%	0.3%	0.06
Pseudo Class	0.7%	1.6%	0.3%	0.0%	0.0%	0.3%	0.3%	0.2%	0.4%	0.4%	0.4%	0.3%	0.0%	1.6%	0.5%	0.06
Sampler	1.2%	3.5%	1.0%	0.0%	0.6%	1.7%	1.0%	0.5%	1.0%	1.0%	1.2%	1.0%	0.0%	3.5%	1.0%	0.10
Restricted Creation	2.3%	0.5%	1.0%	0.0%	0.4%	1.3%	1.5%	1.7%	0.7%	1.5%	1.0%	1.0%	0.0%	2.3%	0.7%	0.14
Coverage	79.5%	79.4%	64.3%	48.0%	83.7%	67.3%	76.9%	76.2%	65.7%	74.9%	71.2%	76.2%	48.0%	83.7%	11.1%	
Entropy	5.27	4.32	4.27	3.32	4.51	4.22	4.74	4.96	4.83	5.08	4.50	4.51	3.32	5.27	0.56	

Table 4: The prevalence, coverage, entropy and marginal entropy of micro patterns in the collections of the pruned corpus.

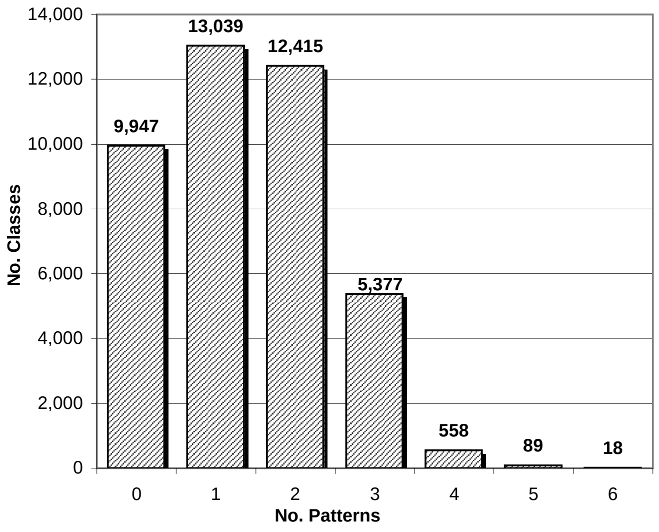


Figure 3: Multiplicity of pattern classification in the classes of the pruned corpus.

classes which matched more than three patterns: 558 classes with four patterns, 89 with five patterns. There were even 18 classes which matched six patterns!

It is interesting to examine some of the classes with multiple classifications. There were 12 classes which matched the same six patterns: Canopy, Restricted Creation, Overrider, Sink, Function Object, and Data Manager. All of these classes are exceedingly similar: they all have a series of pre made instances represented as `public static` fields, a `private` constructor which accepts the name of the created instance (passed as a `String`), a `private` variable to store that name, and a `toString()` method that returns the name of that instance. Here is one of these for example.

```
package javax.xml.rpc;

public class ParameterMode {
   public static final ParameterMode
      IN = new ParameterMode("IN"),
      INOUT = new ParameterMode("INOUT"),
      OUT = new ParameterMode("OUT");

   private String mode;
   private ParameterMode(String mode) {
         this.mode = mode; }
   public String toString() {
         return mode; }
}
```

Interestingly, there is nothing else in all these classes (except for an array of the samples, in the case that the number of samples is large.)

There are 30 classes which match Joiner, Pure Type, Stateless and Sink but no other patterns. Here is one of them for example.

```
package org.freehep.swing.graphics;

public abstract class AbstractPanelArtist
   implements PanelArtist,
      GraphicalSelectionListener {
   public AbstractPanelArtist() { }
}
```

Again, all 30 classes are very similar in structure. They *join* together several empty classes or interfaces, thus helping in enriching the classification hierarchy. In the process, they add a single empty method.

The examples we checked indicated that a multiple patterns match is very precise, yet very narrow. We can think of each pattern combination as a *new* pattern which is more focused than any of its components.

We analyzed the classes where multiple patterns were detected, and found out that there are more than 600 different combinations of multiple patterns (when a combination is a set of patterns detected in a single class). While this number merely provides some vague intuition to the power of the catalog, the entropy measurement can formally describe the catalog's separation power, or: the amount of information that the catalog provides on average. Examining the last row of Tab. 4, we learn that the entropy fluctuates between 4.28 and 5.27. By raising these values to the power of two, we obtain,

CONCLUSION 7.4. *The separation power of the catalog is equivalent to that of a partitioning into 19–39 equal and disjoint sets.*

Marginal Entropy. The last column of Tab. 4 gives the marginal entropy of each of the micro patterns with respect to the entire catalog and all classes in the pruned corpus. In other words, this column specifies the "additional separation power" or added information, when classes which were already matched by the rest of the catalog are matched against this micro pattern.

We see that the marginal entropy of none of the patterns is 0. Therefore, we can state:

CONCLUSION 7.5. *All patterns contribute to the separation power of the catalog.*

In examining the last column we also find that patterns with high prevalence usually exhibit higher marginal entropy, and vice versa, patterns with low prevalence tend to have low marginal entropy. Maximal marginal entropy is achieved by Sink; Implementor follows. In other words, we may argue that Sink contributes the most to the separation power of the catalog. This is in spite of the fact that there are patterns with higher prevalence. In a sense, Sink is more "independent" of the rest of the catalog than other patterns.

The sum of marginal entropies is 5.02, while the entropy of the entire catalog stands at a slightly higher, 5.06. This finding is the basis of our claim that the information brought by the catalog is greater than the sum of its parts.

Variety in Prevalence. Following the table body, there are six columns that give various statistics on the distribution of prevalence of each pattern in the different collections. The first of these columns gives the prevalence of each pattern in the entire (pruned) corpus, i.e., a weighted average of the preceding columns. The two following columns give the (straight) average and median prevalence. Note that in the majority of micro patterns, these three typical values are close to each other. (This situation is typical to symmetrical and normal distributions.)

The next three columns are indicative of the variety in prevalence, giving its minimal and maximal values, as well as the standard deviation. Examining these, we can make the following qualitative conclusion:

CONCLUSION 7.6. *There is a large variety in the prevalence of patterns in different collections.*

For example, Function Object occurs in 24.1% of all classes in JEdit (probably since it is used to realize the COMMAND pattern in this graphic environment), but only in 0.8% of the classes in the Scala compiler. On the other hand, 20.5% of Scala classes match the Pure Type pattern, while the prevalence of this pattern in JEdit is only 2.5%. In JEdit, 37.1% of all classes are instances of Implementor, while only 10.5% of Scala classes match Implementor.

8. Prevalence Differences and Purposefulness

The previous section ended with Conc. 7.6 making the qualitative statement that differences between prevalence levels are "large". In this section, we will make this statement more precise by showing that these differences are *statistically significant*. Concretely, we prove that random fluctuations of prevalence are improbable to generate differences of this magnitude. Thus, we will infer that there exists a non-random mechanism which governs the extent by which patterns are used in different collection.

The statistical validation of Conc. 7.6 can be taken as supporting evidence to our claim that *the patterns in the catalog are purposeful*. One such purpose could be that different software collections serve different needs, and therefore employ different patterns at different levels. Yet another explanation of this non-random process is the difference in programming style and practice between different vendors and their various software teams. We shall discuss these possible explanations in greater detail in the following section.

Statistical Inference. The statistical inference starts by making a *null hypothesis* $\mathcal{H}_0$, by which *patterns are a random property of* JAVA *code*. According to this hypothesis, each pattern has some fixed (yet unknown) probability of occurring in the code, regardless of context or programming style. The number of occurrences of a certain pattern in a collection of n classes is therefore the sum of the n independent random binary variables, one for each class in the collection. The binary variable of a class is 1 precisely when the pattern occurs in that class. If the null hypothesis is true, then changes in prevalence of the pattern across different collections are due to normal fluctuations of the n-sum.

Our objective here is to *reject* the null hypothesis. As usual in statistical inference, we assume $\mathcal{H}_0$ and check the probability that such changes occur under this assumption. More specifically, let $\mathcal{H}_0(p)$ denote the null hypothesis for a pattern p. For each pattern p, we examine the values found in the corresponding row of Tab. 4, i.e, the prevalence level of this pattern in the different collections, and check whether the variety in these can be explained by $\mathcal{H}_0(p)$.

For example, the prevalence of the rarest pattern, Designator, is distributed as follows: 0.0% in two of the collections, 0.1% in two collections, 0.2% in three collections, and 0.3% in the three remaining collections. Are these rather tiny differences which occur in such minuscule prevalence values, meaningful at all?

A precise answer to this question is given by the application of the standard χ^2-test[13] to this row. This test checks whether random fluctuations in prevalence values can give rise, with reasonable probability, to these differences.

Perhaps surprisingly, the test shows that null hypothesis is rejected with confidence level of more than 99%, i.e., $\alpha < 0.01$. (More precisely, the confidence level with Designator 99.75%.) In other words, the probability that the changes in the prevalence level of Designator can be explained by $\mathcal{H}_0(p)$ is less than 0.01.

In applying the test to each of the patterns we find that hypothesis $\mathcal{H}_0(p)$ is rejected with confidence level of 99.9%, i.e., $\alpha < 0.001$ for all the patterns in the collection, with only two exceptions: Designator, for which the confidence level is 99.75%, and Cobol Like, in which the confidence level is 96%.

At the usual 95% confidence level employed in statistical inference, the differences in prevalence level of Cobol Like are statistically significant. It is tempting to declare that $\mathcal{H}_0(p)$ is rejected for *all* p. We shall however use a fixed confidence level of 99%, i.e., declare that the null hypothesis is rejected only if $\alpha < 0.01$. The following argument explains why.

> However, even if the null hypothesis is true, it is *expected* that in about 5% of all cases, the 95% of this confidence level will be reached. Since we used a battery of 27 tests for all $\mathcal{H}_0(p)$, the expected number of times in which this confidence level is reached, is greater than 1.

[13] Read "Chi-squared-test".

CONCLUSION 8.1. *With the exception of* Cobol Like*, changes in prevalence of each of the micro patterns in the collections of the pruned corpus are significant.*

Pair-wise Separation. The above conclusion does not provide means of *understanding* the nature of the changes. It merely says that these changes as a whole are (statistically) significant. Furthermore, the rejection of $\mathcal{H}_0(p)$ does not mean that *every change in prevalence level of each pattern in any two collections* is significant.

Conc. 8.1 only says that *not all* changes in the collections are a matter of coincidence. Despite the great variety, some patterns exhibit the same prevalence level in different collections. For example, the prevalence of State Machine in Tomcat and Poseidon is almost the same (round 1.7%); its (rounded) prevalence in Scala and Shared is 1.8%. Is each of these differences significant?

Let $\mathcal{H}_0[c_1, c_2](p)$ be the null hypothesis that the prevalence of a pattern p in collections c_1 and c_2 is the same. To check this hypothesis, we apply a χ^2-test to determine whether the difference in proportions (i.e., a "single degree freedom", in χ^2-test) in the two collections is significant. The test result is that hypothesis $\mathcal{H}_0[\text{Tomcat}, \text{Poseidon}])(\text{State Machine})$ cannot be rejected by the test. The test similarly fails to reject the hypothesis

$$\mathcal{H}_0[\text{Scala}, \text{Shared}](\text{State Machine}).$$

We say that pattern p *separates* the collections c_1 and c_2 if

$$\mathcal{H}_0[c_1, c_2)](p)$$

is rejected. The application of a χ^2-test gives us an effective means for concentrating only on the significant differences in prevalence levels.

Corpus Separation Index. Let us now define a metric of the average extent by which a pattern distinguishes between different collections.

DEFINITION 5. *Assume some fixed confidence level. Let $\mathcal{C}$ be a software corpus. Let p be a pattern. Then, the notation $\Upsilon(p, \mathcal{C})$ (or for short just $\Upsilon(p)$ when $\mathcal{C}$ is clear from context), stands for the* separation index of p *(with respect to $\mathcal{C}$) is the fraction of rejected null hypotheses $\mathcal{H}_0[c_1, c_2](p)$ (at the fixed confidence level) out of all such hypotheses where c_1 and c_2 vary over $\mathcal{C}$, $c_1 \neq c_2$.*

The separation index becomes useful because the χ^2-test is sensitive to outliers: Suppose that the prevalence of a pattern p in a single collection $c \in \mathcal{C}$ is distant from the average prevalence, while the prevalence in the other collections in $\mathcal{C}$ is very close to the average prevalence. Then, the test will reject the null hypothesis $\mathcal{H}_0(p)$. In contrast, $\mathcal{H}_0[c_1, c_2](p)$ will be rejected only if $c_1 = c$ or $c_2 = c$.

Low separation index of a pattern indicates that the pattern prevalence is more stable in different collections.

Fig. 4 shows the separation index of the patterns in the catalog with respect to the pruned corpus (black columns) and the JRE corpus (white columns).

Not surprisingly, the minimal value is that of Cobol Like, which separates only 2 out of the 36 pairs of the pruned corpus **Pruned**

$$\Upsilon(\text{Cobol Like}, \textbf{Pruned}) = 5.6\%.$$

It is followed by $\Upsilon(\text{Designator}) = 8.33\%$. The highest separation index, 89%, is achieved by Function Object, where the second highest value is $\Upsilon(\text{Overrider}) = 86\%$. The median is 44%, while the average separation index is 47%.

CONCLUSION 8.2. *The difference in prevalence levels between two collections is significant in one out of two cases.*

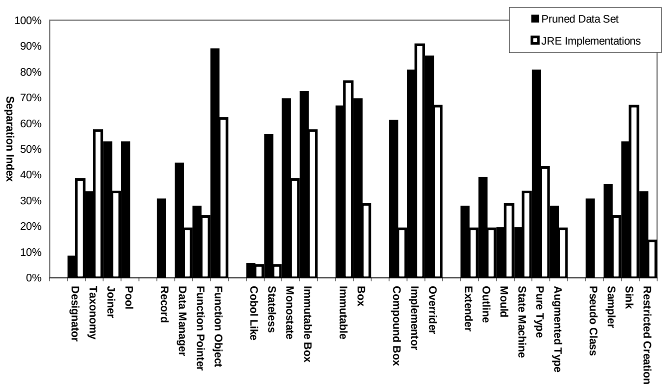


Figure 4: The separation index of the patterns with respect to the pruned corpus and the different implementations of the JRE ($\alpha < 0.01$).

Conclusions Conc. 8.1 and Conc. 8.2 together are the statistically sound counterpart of the qualitative statement in Conc. 7.6.

9. Intermediate Discussion

The previous section established the significance of the variety in prevalence level of the same pattern in different collections. In this section, we shall discuss several ways of interpreting this significance.

We first note that the corpus size makes it possible to establish the significance of even relatively small differences in prevalence levels. But, the nature of the statistical tests we employ is that they take an appropriate account of both the corpus size but also of the expected prevalence. Even with large data sets, not every difference in prevalence level is significant:

- Consider the difference in prevalence level of an individual pattern between two specific collections. Fig. 4 shows that in only about half of the cases there was significance to the difference.

- As mentioned in Sec. 6 we tried to expand the corpus with three other ports of Sun's implementation of the JRE, due to IBM, HP, and BEA. As it turned out, the differences in prevalence level of these ports were not statistically significant.

We can think of at least five different phenomena which can explain, alone or together, the findings of Sec. 8

1. *Requirement Variety.* The different collections serve different needs, which call for different patterns.

2. *Style Variety.* The different collections are implemented by different vendors employing different programming policies, styles, and individuals, all reflected by patterns prevalence.

3. *Replication.* We know that programmers tend, or are at least encouraged, to reuse both design and code. Programmers may copy classes, changing only a few lines of codes, instead of factoring out similarities[14]. If this happens often, the number of "independent" classes in a collection is smaller than the actual number of classes. A random fluctuation in a pattern prevalence is amplified by this replication, and interpreted to be significant even if it is not so.

[14]In some cases, code duplication cannot be avoided due to the absence of advanced abstraction mechanisms, (such as multiple inheritance, mixins, anonymous functions, and traits) in the language.

4. *Population Contamination* Our experiments cannot tell a difference in prevalence level is a result of a moderate global change to the entire population, or of an accentuated change to a subpopulation.

   To understand this better, let us assume that Common State occurs naturally in JAVA code with probability of 7%. Then, if we take a set of $10,000$ classes, about 700 of these will be a Common State. Finding instead 767 classes, is, so tells us the χ^2-test, statistically significant. What the test fails to say is whether the increase in the number of occurrences is a result an increased global tendency to use Common State, or of (say) having a sup-population of 350 classes, whose specific domain is such that the prevalence of Common State is 26%.

5. *Dormant Abstraction.* It could be the case that the micro-patterns found here are a reflection of higher level, *deep* patterns which are still not known to us. The difference in prevalence of micro patterns could be a reflection of difference in prevalence of the "deep patterns", which capture the "true" differences between collections.

We would like to attribute changes in the use of patterns primarily to requirement variety, and only then to style variety. But, these changes could be a result of code replication, or population contamination. These two explanations represent in fact the two faces of the same phenomena, i.e., that different classes are not independent of each other. Finally, dormant abstraction may mean that we are examining the wrong patterns.

Statistical inference cannot positively *confirm* any of these explanations. It can however, be employed for the rejection of one or more such conjectures, and for estimating the relative contribution of the factors which are not rejected. Such an investigations requires carefully designed experiments, and lies out of scope of this work.

In an initial experimentation with a bunch of "pseudo-patterns", which are not expected to carry any purpose, we made some interesting discoveries.

- Pseudo-patterns computed by hashing the class pool into a single bit showed, at times, significance, although not as strong as we found for micro patterns. This finding indicates that the extent of code replication in the corpus is small, but probably measurable.

- The statistical tests can trace in the corpus more than design information. For example, the use of a code obfuscator in parts of Poseidon, generated short named classes, which made significant changes to the prevalence of a "non-sense" pattern occurring whenever the length of the class name is a prime number. The dormant abstraction of naming convention could be detected by significant changes to the same "pattern".

- We were able to find dormant abstraction, of (so we guess) our patterns, in examining classes with exactly one method, and no instance fields. In other meaningless patterns, e.g., requiring that a class has precisely two methods and two instance fields, significance was found.

10. The Evolution of Software Collections

We now turn to the quest of checking the persistence of micro patterns across different implementations of the same design, and in the course of the software life cycle. To this end we consider the seven different implementations of the JRE as discussed in Sec. 6.

10.1 Prevalence in JRE Implementations

Tab. 5 is structured similarly to Tab. 4 except that in Tab. 5 we compare the micro pattern prevalence in the seven implementations of the JRE, i.e., in the corpus defined by

$$\textbf{JRE} = \{\mathsf{Kaffe}^{1.1}, \mathsf{Kaffe}^{1.1.4}, \mathsf{Sun}^{1.1}, \mathsf{Sun}^{1.2}, \mathsf{Sun}^{1.3}, \mathsf{Sun}^{1.4.1}, \mathsf{Sun}^{1.4.2}\}$$

Comparing the two tables we see that the values in the average, total, and median lines in Tab. 5 are close, just as they are in Tab. 4.

In comparing the standard deviation column (σ) in the two tables, we see that the variety in coverage level and entropy is much smaller in the related collections (Tab. 5) than the variety in the related collections (Tab. 4). For the majority of patterns (18 out of the 27), the variety in prevalence level in Tab. 5 is smaller than the variety in Tab. 5.

> The variety of four patterns is about the same in both corpora. Only five patterns, Designator, Taxonomy, State Machine, Immutable and Sink showed a greater variety in the JRE-collections than in the unrelated collections.
>
> Examining these patterns, we see that there was a large drop in their prevalence level with the progress of JRE implementations.
>
> The drop in Immutable is explained by a change in the root of the exceptions hierarchy of JRE, `Throwable`, which broke the immutability of all of the classes in it.
>
> The drop in Designator, Taxonomy, State Machine and Sink, is not so much in relative numbers but rather due to the fact that the development of new branches of the standard library did not make much new use of these patterns. In particular, the introduction of the fairly large and complex Swing library in $\mathsf{Sun}^{1.2}$, has induced a corresponding decrease in the ratio of Sink classes.

We can therefore make the following qualitative conclusion:

CONCLUSION 10.1. *Pattern prevalence tends to be the same in software collections which serve similar purposes, independent of the size of the collection.*

Note that the two most largest differences are 19% in Implementor, between $\mathsf{Sun}^{1.1}$ and $\mathsf{Sun}^{1.2}$, and an 11% drop in Immutable between $\mathsf{Sun}^{1.3}$ and $\mathsf{Sun}^{1.4.1}$. The first difference can be attributed to the introduction of large interface-based libraries in $\mathsf{Sun}^{1.2}$ (such as the *Swing* library and the `sun.java2d.*` packages). The latter difference is, as explained above, due to the change of class `Throwable` in $\mathsf{Sun}^{1.4.1}$.

To appreciate the greater similarity in prevalence values, we can recheck the null hypothesis $\mathcal{H}_0[p]$. This time with respect to the collections in the **JRE** corpus. As it turns out, the hypothesis cannot be rejected as often as in the pruned corpus. The difference in the prevalence levels of Cobol Like were insignificant here just as in the pruned corpus, but there were five additional patterns for which the differences in prevalence levels not significant: Outline, Augmented Type, Pseudo Class, Pool, Stateless, and Record.

10.2 Proximity of JRE Implementations

The finding that $\mathcal{H}_0[p]$ is rejected less often with JRE implementations, supports the qualitative statement in Conc. 10.1. But, in order to make the conclusion more precise, we need a sound statistical method of comparing the variation in pattern prevalence among the related collections, i.e., corpus **JRE**, comprising the different implementations of the JRE, and unrelated collections, i.e., corpus

Collection	Kaffe$^{1.1}$	Kaffe$^{1.1.4}$	Sun$^{1.1}$	Sun$^{1.2}$	Sun$^{1.3}$	Sun$^{1.4.1}$	Sun$^{1.4.2}$	Total	Average	Median	Min	Max	σ
Designator	1.3%	0.8%	0.8%	0.4%	0.4%	0.3%	0.2%	0.4%	0.6%	0.4%	0.2%	1.3%	0.4%
Taxonomy	11.0%	5.6%	13.8%	6.7%	5.8%	5.1%	4.4%	5.9%	7.5%	5.8%	4.4%	13.8%	3.5%
Joiner	0.4%	0.3%	0.0%	0.8%	0.9%	1.2%	0.7%	0.8%	0.6%	0.7%	0.0%	1.2%	0.4%
Pool	1.9%	1.9%	1.4%	1.6%	1.8%	2.3%	1.9%	1.9%	1.8%	1.9%	1.4%	2.3%	0.3%
Sink	21.9%	27.5%	23.5%	17.0%	17.6%	17.3%	20.6%	19.4%	20.8%	20.6%	17.0%	27.5%	3.9%
Record	0.2%	0.2%	0.5%	0.5%	0.6%	0.6%	0.4%	0.5%	0.4%	0.5%	0.2%	0.6%	0.2%
Data Manager	2.7%	2.8%	2.3%	1.5%	1.9%	1.9%	1.8%	1.9%	2.1%	1.9%	1.5%	2.8%	0.5%
Function Pointer	1.0%	1.1%	0.5%	2.1%	1.2%	1.7%	2.0%	1.6%	1.4%	1.2%	0.5%	2.1%	0.6%
Function Object	1.1%	1.2%	1.8%	8.0%	7.7%	6.9%	7.7%	6.5%	4.9%	6.9%	1.1%	8.0%	3.3%
Cobol Like	0.9%	0.6%	0.5%	0.4%	0.4%	0.4%	0.4%	0.4%	0.5%	0.4%	0.4%	0.9%	0.2%
Stateless	9.0%	8.9%	7.0%	9.3%	8.6%	9.5%	9.8%	9.2%	8.9%	9.0%	7.0%	9.8%	0.9%
Common State	2.0%	1.3%	1.9%	1.4%	2.8%	3.6%	2.4%	2.5%	2.2%	2.0%	1.3%	3.6%	0.8%
Canopy	5.1%	4.8%	2.9%	10.2%	9.1%	9.1%	9.8%	8.7%	7.3%	9.1%	2.9%	10.2%	2.9%
Immutable	13.3%	5.1%	15.1%	17.6%	16.7%	6.8%	7.6%	10.7%	11.7%	13.3%	5.1%	17.6%	5.2%
Box	8.0%	4.1%	4.7%	4.5%	4.6%	5.3%	4.6%	4.9%	5.1%	4.6%	4.1%	8.0%	1.3%
Compound Box	7.0%	6.0%	8.4%	5.4%	5.5%	5.7%	6.0%	5.9%	6.3%	6.0%	5.4%	8.4%	1.1%
Implementor	9.7%	17.0%	9.3%	27.6%	27.1%	21.5%	26.0%	23.2%	19.7%	21.5%	9.3%	27.6%	7.9%
Overrider	6.6%	5.7%	7.7%	9.8%	9.2%	12.3%	12.4%	10.5%	9.1%	9.2%	5.7%	12.4%	2.6%
Extender	6.5%	4.9%	4.5%	4.1%	4.1%	4.1%	4.3%	4.3%	4.6%	4.3%	4.1%	6.5%	0.9%
Outline	1.9%	2.8%	2.3%	1.7%	1.7%	1.8%	1.8%	1.9%	2.0%	1.8%	1.7%	2.8%	0.4%
Trait	1.6%	2.3%	1.4%	1.9%	1.9%	1.3%	1.3%	1.6%	1.7%	1.6%	1.3%	2.3%	0.4%
State Machine	1.6%	3.1%	2.5%	1.3%	1.4%	1.8%	1.5%	1.7%	1.9%	1.6%	1.3%	3.1%	0.7%
Pure Type	10.1%	14.9%	12.4%	8.4%	8.7%	9.5%	7.7%	9.3%	10.2%	9.5%	7.7%	14.9%	2.6%
Augmented Type	0.9%	1.3%	1.0%	0.6%	0.7%	0.7%	0.6%	0.7%	0.8%	0.7%	0.6%	1.3%	0.3%
Pseudo Class	1.1%	1.2%	0.5%	1.1%	1.0%	0.8%	0.7%	0.9%	0.9%	1.0%	0.5%	1.2%	0.3%
Sampler	2.5%	1.2%	1.6%	1.3%	1.2%	1.1%	1.2%	1.3%	1.5%	1.2%	1.1%	2.5%	0.5%
Restricted Creation	2.0%	2.5%	1.3%	1.2%	1.8%	2.2%	2.3%	2.0%	1.9%	2.0%	1.2%	2.5%	0.5%
Coverage	74.8%	82.9%	73.3%	79.5%	80.3%	79.5%	79.5%	79.5%	78.5%	79.5%	73.3%	82.9%	3.3%
Entropy	4.98	5.08	4.68	5.17	5.25	5.25	5.27	5.34	5.10	5.17	4.68	5.27	0.21

Table 5: The prevalence, coverage and entropy of micro patterns in different implementations of the JRE.

Pruned. To this end, we shall use the *separation index* of a pattern (Def. 5).

We expect that patterns will distinguish between more pairs of collections in the unrelated applications. Conversely, since we believe that the collections in **JRE** are relatively similar to each other, we expect a lesser ability of patterns to distinguish between these collections.

Fig. 4 helps in visualizing the meaning of this expectation. The black columns in the figure, represent the separation indices of the patterns with respect to the pruned corpus, while the white columns represent the separation indices of the same patterns with respect to the JRE implementations. The question is therefore: "*Are the black columns significantly taller than the white columns?*"

We know that the average separation index with respect to the pruned corpus is 47%, while the average with respect to the JRE's is 33%. But we cannot tell whether this difference of 14% is significant without determining the distribution of the separation index.

Let $\mathcal{H}'_0[\textbf{Pruned}, \textbf{JRE}]$ be the null hypothesis that the patterns have the same ability to distinguish between collections in both corpora. Then, under this assumption, the separation index will have the same (unknown) distribution in both sets, and the event

$$\Upsilon(p, \textbf{Pruned}) < \Upsilon(p, \textbf{JRE}) \tag{2}$$

will be just as likely as the event

$$\Upsilon(p, \textbf{Pruned}) > \Upsilon(p, \textbf{JRE}). \tag{3}$$

We therefore have that $\mathcal{H}'_0[\textbf{Pruned}, \textbf{JRE}]$ implies that n_e, the number of patterns in which (2) holds follows a binomial distribution of 27 tosses of a balanced coin.

Doing the count in Fig. 4 we obtain $n_e = 7$. A standard (one-sided) test of this distribution reveals that the hypothesis

$$\mathcal{H}'_0[\textbf{Pruned}, \textbf{JRE}]$$

is rejected at confidence level 99%.

We therefore obtain the following statistically sound counterpart of Conc. 10.1.

CONCLUSION 10.2. *With high confidence value, micro patterns tend to have more similar prevalence values in different implementations of the JRE than the pruned corpus.*

10.3 Progressive JRE Implementations

We can employ the same statistical test to check whether patterns tend to exhibit greater similarity in their prevalence levels in progress versions of the JRE, than in less related versions. To this end we used a smaller corpus **SUN**, comprising the four last versions of Sun's JRE implementations, i.e.,

$$\textbf{SUN} = \{\text{Sun}^{1.2}, \text{Sun}^{1.3}, \text{Sun}^{1.4.1}, \text{Sun}^{1.4.2}\},$$

and compared the separation indices with respect to it, with the separation indices with respect to the entire corpus of **JRE**s .

At it turns out, there were 20 patterns p in which

$$\Upsilon(p, \textbf{SUN}) < \Upsilon(p, \textbf{JRE}). \tag{4}$$

By checking the same binomial distribution as before, we obtain that hypothesis $\mathcal{H}'_0[\textbf{SUN}, \textbf{JRE}]$ is rejected at the 99% confidence level.

We can thus strengthen Conc. 10.2 by the following.

CONCLUSION 10.3. *With high confidence level, progressive versions of the JRE, tend to exhibit more similar pattern prevalence levels than the entire range of implementations of the JRE.*

It is even possible to use the patterns catalog to examine the proximity of *individual* JRE implementations. To do so, consider the random variable representing the prevalence level of each of the patterns. Tab. 6 presents the values of $r(c_1, c_2)$, the Pearson correlation between the values of this variable in all pairs c_1, c_2 of implementations of the JRE.

A qualitative inspection of the table reveals that all correlation values are high. The smallest correlation, 0.69 is between the first and the last of Sun's implementation of the JRE.

Examining the last row of the table, we see that the correlation of Sun$^{1.4.2}$ with prior editions is increasing with version number. By

	$\text{Kaffe}^{1.1}$	$\text{Kaffe}^{1.1.4}$	$\text{Sun}^{1.1}$	$\text{Sun}^{1.2}$	$\text{Sun}^{1.3}$	$\text{Sun}^{1.4.1}$	$\text{Sun}^{1.4.2}$
$\text{Kaffe}^{1.1}$	1.00	0.87	0.97	0.75	0.76	0.76	0.74
$\text{Kaffe}^{1.1.4}$	0.87	1.00	0.85	0.76	0.78	0.85	0.85
$\text{Sun}^{1.1}$	0.97	0.85	1.00	0.73	0.74	0.72	0.69
$\text{Sun}^{1.2}$	0.75	0.76	0.73	1.00	1.00	0.93	0.94
$\text{Sun}^{1.3}$	0.76	0.78	0.74	1.00	1.00	0.94	0.95
$\text{Sun}^{1.4.1}$	0.76	0.85	0.72	0.93	0.94	1.00	0.99
$\text{Sun}^{1.4.2}$	0.74	0.85	0.69	0.94	0.95	0.99	1.00

Table 6: The Pearson correlation between patterns prevalence level of patterns in different implementations of the JRE; all values are significant at the $\alpha < 0.01$ confidence level.

applying the standard technique of Fisher's transformation, we can even check whether these increases are statistically significant.

As it turns out, the increase from $r(\text{Sun}^{1.4.2}, \text{Sun}^{1.1}) = 0.69$ to $r(\text{Sun}^{1.4.2}, \text{Sun}^{1.2}) = 0.94$ is significant, and so is the increase from $r(\text{Sun}^{1.4.2}, \text{Sun}^{1.3}) = 0.95$ to $r(\text{Sun}^{1.4.2}, \text{Sun}^{1.4.1}) = 0.99$.

On the other hand, the difference between $r(\text{Sun}^{1.4.2}, \text{Sun}^{1.2}) = 0.94$ and $r(\text{Sun}^{1.4.2}, \text{Sun}^{1.3}) = 0.95$ is statistically insignificant. These findings strengthen Conc. 10.3.

10.4 Programming Style vs. Specification

Tab. 6 includes an interesting case which can be used to compare the contribution to the choice of micro pattern of programming style with that of software specification.

Consider collections $\text{Sun}^{1.1}$ and $\text{Kaffe}^{1.1}$ which represent two independent implementations of almost identical specifications. We see a very high correlation value, $r(\text{Kaffe}^{1.1}, \text{Sun}^{1.1}) = 0.97$, between the two collections.

Let us now examine the correlation values of $r(\text{Sun}^{1.2}, \text{Sun}^{1.1})$ and $r(\text{Kaffe}^{1.1}, \text{Kaffe}^{1.1.4})$ which record similarity in pattern prevalence in cases that the specification changed (mostly by expanded functionality), but the programming style and Vendor culture presumably did not change so much. We have:

$$\begin{aligned} r(\text{Kaffe}^{1.1}, \text{Kaffe}^{1.1.4}) = 0.87 < r(\text{Kaffe}^{1.1}, \text{Sun}^{1.1}) \\ r(\text{Sun}^{1.2}, \text{Sun}^{1.1}) = 0.73 < r(\text{Kaffe}^{1.1}, \text{Sun}^{1.1}) \end{aligned} \quad (5)$$

Moreover, in applying the Fisher transformation to the respective r values, we find that both inequalities in (5) are statically significant. The finding in this test case suggests that micro pattern application tends to be determined by the specification more than the programming style.

Comments. First, note that the conclusions in this section do not mean that micro patterns are the only means, or even the most effective tool, for determining proximity of software collections.

We expect that many other metrics, including metrics derived from non-purposeful patterns (similar to the pseudo patterns mentioned in Sec. 9) will exhibit similar distinguishing capabilities. What we have established is that micro patterns are not a random property of code, and that its behavior in the course of changes in specification is in accordance with our natural understanding of these.

Second, note that the single test case presented in Sec. 10.4 indicates that specification has more impact than style, but it cannot prove such a point. It is difficult to run a controlled experiment of this sort. The costs of software development make the independent implementation of the same specification a true rarity.

Preservation of Prevalence Level. It is tempting to think that conclusions 10.1–10.3 are a result of the fact that the same classes occur in all implementations collections. This is not true, since the numbers of classes in each such implementation is very different; the series of Sun's JRE version exhibits dramatic increase in the number of classes, mostly due to considerable functionality added at each new version. The *ratio* of classes which use certain patterns is preserved. This means that the *added* functionality is implemented in a fashion which is similar to the existing functionality.

Independent Implementation of the Same Class. It turns out also that the different implementations of the same class are not necessarily with the same pattern. To measure the tendency of implementing a specification with the same pattern, we considered for each pair of implementations, the Pearson correlation between the events of implementing a specific class with the same pattern in the two implementations. A total of $27 \cdot 7 \cdot 6/2 = 567$ correlation values were thus computed. (Except for the small number of exceptions discussed below, all these values were significant at the 99.9% confidence level or higher.)

As expected, there were no cases of negative correlation. Moreover, in 90% of all cases, the correlation was greater than 0.6; in 46% of the cases, it was greater than 0.9. Since the implementation was carried by two independent vendors, we can conclude.

CONCLUSION 10.4. *Independent implementations of the same specification have a strong tendency to use the same pattern.*

It should be stated however that in only 65 cases, the correlation was 1, i.e., it was the case that a class used a specific pattern in one implementation if and only if it used the same pattern in the other implementation.

Of course, we do not know if all cases in which implementations of the same class *did not* chose the same patterns are a result of consciousness design decision. These cases could also be an artifact of inaccuracies of our automatic pattern detection tool or of our pattern definition.

It may take a moment's thought to be convinced that Conc. 10.1 and Conc. 10.4 do not contradict Conc. 8.1. What we have in fact is the following observation:

CONCLUSION 10.5. *Although the prevalence level tends to be similar in similar implementations, the small changes in the prevalence level between any two implementation of the JRE (across all patterns) are statistically significant.*

11. Related Work

Cohen and Gil [13] supplied some statistical evidence to the existence of *common programming practice*, which "good" programmers will follow in their coding. Their conclusions were obtained from a set of simple metrics, such as: number of parameters of a method, bytecode size of a method, number of static method calls, etc. Given the somewhat "technical" nature of these metrics, the deduction of meaningful conclusions regarding the design of a program, from a given vector of metrics values, is not an easy task.

In this paper, we took the natural challenge of bridging the gap: finding micro patterns which are at a slightly higher level than e.g., the number of parameters to a method, but at a lower level than design patterns.

Van Emde Boas [45] describes the trade off of expressivity (of the language used for describing design patterns) vs. the complexity of the pattern detection problem. He showed that lack of syntactic constraints on the design pattern definitions, results in the detection problem being undecidable.

Kraemer and Prechelt [36] developed the Pat system which detects structural design patterns (ADAPTER, BRIDGE, COMPOSITE, DECORATOR, PROXY) by inspecting a given set of C++ header files (`.h`), and storing extracted data as *Prolog* facts. Identification is carried out by invoking a Prolog query against a set of predefined

Prolog rules describing the identifiable design patterns. This system had a detection precision of 14 – 50%. As the authors claim, the precision can be significantly improved by checking method call delegation information, which cannot be obtained from header files. Our approach is expected to yield better results:

- The richer information available at `.class` file will allow our tools to inspect method call delegations, detect more types of patterns, and reduce the number of false positives.

- We are not restricting our research to Gamma et al.'s [22] design patterns. Any micro pattern is applicable.

Heuzeroth et al. [25] combine static and dynamic analysis for detection of design patterns (Behavioral: OBSERVER, MEDIATOR, CHAIN OF RESPONSIBILITY, VISITOR; Structural: COMPOSITE) in JAVA applications. The static analyzer applies various predicates over the source code (`.java` files) to obtain a set of candidates. The dynamic analyzer employs code instrumentation techniques to trace the behavior of the candidates at runtime. A candidates whose behavior is not conforming with the expected behavior of the relevant design pattern is filtered out. This technique's dependency on runtime information is a major drawback.

Brown [9] uses dynamic analysis of *Smalltalk* programs for the detection of Gamma et al. patterns. His technique is based on tracing of messages sent between objects.

The need for enhanced documentation tools has been stated by several works in the area of software visualization [19, 41, 42]. Another similar research is the work of Lanza and Ducasse [26], which suggest a technique for classifying methods of *Smalltalk* classes to one of five categories by inspecting their implementation. These authors' classification algorithm is partially based on common naming conventions.

Micro patterns are related also to a number of systems which allowed the programmer to add auxiliary, automatically checkable rules to code. Examples include Minsky's *Law Governed Regularities* [32, 33], or Aldrich, Kostadinov and Chambers's work on alias annotation [2]. Micro patterns are different in that they restrict the programmer's freedom in choosing such rules (unless the user comes with a new pattern), but on the other hand give a set of simple, pre-made, well defined rules backed up by extensive empirical support.

Statistical inference in the context of software was used in the past. For example, Soloway, Bonar, and Ehrlich [40], employed the χ^2-test to answer questions such as the extent by which advanced programmers have greater tendency to use certain programming idioms, and the extent by which language support for the preferred idioms promotes program correctness.

12. Discussion and Further Research

People use patterns without thinking. This phenomenon is a consequence of the recognition built into every one of us, that *routine* is easier and safer than the time consuming and error-prone process of *decision making*. As demonstrated so many times in the past, patterns exist also in the programming world. In languages with rich system of attributes such as JAVA it is clear there are many (statistical) correlations between these attributes. For example, we expect classes which define static fields to define static methods, etc.

Micro patterns step further beyond the simple conclusion that there are many inter-correlations between the setting of (say) attributes and selection of types. In this paper we showed, probably for the first time, that there are distinct patterns which the majority of JAVA software follows. In fact, we gave meaning, name and significance to many of these correlations.

We described a catalog of micro patterns, which can be used as a mental skeleton to mold mundane modules, allowing programmers to become more productive. For example, by using the catalog, much of the coding work is reduced to the mere issue of selecting a pattern for a class (often dictated by the system design), and then laboriously filling in the missing details.

We showed (Conc. 7.1) that an overwhelming majority of JAVA classes follows one or more of the patterns in the catalog. (The remaining classes either fit yet unknown patterns, or represent code locations which required more skill than routine.) We used several statistical methods to increase the confidence that these patterns capture sound ideas. For example, we gave statistical evidence to the claim that independent implementations of the same specification tends to use the same pattern and that this choice is preserved in the course of software evolution (Conc. 10.4).

Despite the fact that more than half the classes can be described by one of the five leading patterns (Conc. 7.3), we found that each of the patterns in the catalog contributes (Conc. 7.5) to the 4–5 bits or so of design information that the catalog as a whole reveals (Conc. 7.4).

After noticing (Conc. 7.6) that there is a considerable variety in the use of patterns in different domains, statistical analysis was carried out to understand better the nature of this variety. This analysis has shown that in almost half of the cases, changes in a pattern's prevalence levels, between two software collections, were not an artifact of random fluctuation. This indicates that the choice of patterns is not merely a follow up of the language constraints and that it is effective for distinguishing programming context.

We are currently engaged in devising a specification language, which based on FOPL, will make it possible to concisely and precisely define patterns. Within this langauge framework, it is interesting to add weights to part of the definitions, which will make it possible to measure the proximity of a class to a pattern. Weights should also make it possible to build systems which not only discover the use of micro patterns, but also help the user correct his software—by offering concrete recommendations of how to make certain classes a better match to the acquired knowledge base.

With the development of automatic tools for tracing patterns, and the evidence of their significance, it is possible and interesting to expand the notion of micro patterns by studying kinds of interactions between classes obeying various micro patterns and even developing patterns to specify sorts of such interaction. A typical question of this sort is whether base class category is indeed used more as inheritance basis. Such a research direction may even mature to tools making more global advice. For example, in a hierarchy where a Pure Type class is subclassed by several Implementor classes, the root class can possibly be turned into a Trait or an Outline class, thus capturing some of the similarities of its subclasses.

On the other hand, we expect that nano-patterns, i.e., patterns of methods can be defined and traced in the code, and that the combination of micro patterns and nano-patterns will be a better aid to design, documentation and software comprehension.

Also interesting is the subjective value of patterns, i.e., the measurable extent by which they improve (or degrade) practitioners productivity and sense of empowerment.

Finally, perhaps the most challenging work is in discovering concrete connections between micro patterns and software quality.

Acknowledgements. The authors pay tribute to Kent Beck for his wise comments which greatly helped shaping this paper. The meticulous reading and the comments of the anonymous reviewers of the OOPSLA 2005 Program Committee are happily acknowledged. In particular, we thank the referees for bringing us to the observation that statistical significance can be employed to better understand a dormant purpose.

Special thanks go to James Noble, for his deep and inspiring input. The use of the term software corpus also follows his suggestions.

13. REFERENCES

[1] E. Agerbo and A. Cornils. How to preserve the benefits of design patterns. In *Proc. of the 13th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'98)*, pages 134–143, Vancouver, British Columbia, Oct.18-22 1998. ACM SIGPLAN Notices.

[2] J. Aldrich, V. Kostadinov, and C. Chambers. Alias annotations for program understanding. In *Proc. of the 17th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'02)*, Seattle, Washington, Nov. 4–8 2002. ACM SIGPLAN Notices.

[3] P. Arnold and J. Gosling. *The Java Programming Language*. The Java Series. Addison-Wesley, 1996.

[4] K. Beck. *Smalltalk: best practice patterns*. Prentice-Hall, 1st edition, 1997.

[5] K. Beck. *JUnit Pocket Guide*. O'Reilly, 2004.

[6] A. Blewitt, A. Bundy, and I. Stark. Automatic verification of Java design patterns. In *Proc. of the 16th IEEE Conference on Automated Software Engineering (ASE'01)*, pages 324–327, San Diego, California, 2001. IEEE Comp.

[7] J. Bloch. *Effective Java Programming Language Guide*. Addison-Wesley, 1st edition, June 2001.

[8] G. Bracha and W. R. Cook. Mixin-based inheritance. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications / Proceedings of the European Conference on Object-Oriented Programming (OOPSLA/ECOOP'90)*, pages 303–311, Ottawa, Canada, Oct. 21-25 1990. ACM SIGPLAN Notices.

[9] K. Brown. *Design Reverse-Engineering and Automated Design Pattern Detection in Smalltalk*. Masters thesis, North Carolina State University, 1996.

[10] M. Campione, K. Walrath, and A. Huml. *The Java Tutorial: A Short Course on the Basics*. Addison-Wesley, 2000.

[11] D. N. Card and R. L. Glass. *Measuring software design quality*. Prentice-Hall, 1990.

[12] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, June 1994.

[13] T. Cohen and J. Gil. Self-calibration of metrics of Java methods. In *Proc. of the 37th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS'00 Pacific Conference)*, pages 94–106, Sydney, Australia, Nov. 20-23 2000. Prentice-Hall.

[14] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1st edition, June 1990.

[15] B. Demsky and M. C. Rinard. Automatic detection and repair of errors in data structures. In *Proc. of the 18th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'03)*, pages 78–95, Anaheim, California, USA, Oct. 2003. ACM Press.

[16] A. H. Eden. Formal specification of object-oriented design. In *Proc. of the International Conference on Multidisciplinary Design in Engineering (CSME-MDE'01)*, Montreal, Canada, Nov. 21-22 2001.

[17] A. H. Eden. A visual formalism for object-oriented architecture. In *Proc. of the 6thIntegrated Design and Process Technology (IDPT'02)*, California, June 23-28 2002. Society for Design and Process Science.

[18] A. H. Eden and R. Kazman. Architecture, design, implementation. In *Proc. of the 25th International Conference on Software Engineering (ICSE'03)*, pages 149–159, Portland, Oregon, May 3-10 2003. IEEE Comp.

[19] S. G. Eick, J. L. Steffen, and E. E. J. Sumner. Seesoft-a tool for visualizing line oriented software statistics. *IEEE Trans. on Soft. Eng.*, 18(11):957–968, Nov. 1992.

[20] G. Florijn, M. Meijers, and P. van Winsen. Tool support for object-oriented patterns. In *Proc. of the 11th European Conference on Object-Oriented Programming (ECOOP'97)*, pages 472–495, Jyväskylä, Finland, June 9-13 1997. Springer.

[21] E. Gamma, R. Helm, R. E. Johnson, and J. M. Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In *Proc. of the 7th European Conference on Object-Oriented Programming (ECOOP'93)*, pages 406–431, Kaiserslautern, Germany, July 26-30 1993. Springer.

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

[23] A. Goldberg. *Smalltalk-80: The Interactive Programming Environment*. Addison-Wesley, 1984.

[24] M. Grand. *Patterns in Java: A Catalog of Reusable Design Patterns Illustrated with Uml,Volume 1*. Wiley, 2002.

[25] D. Heuzeroth, T. Holl, G. Höström, and W. Löwe. Automatic design pattern detection. In *Proc. of the 11thInternational Workshop on Program Comprehension (IWPC'03)*, page 94, Portland, Oregon, USA, May 2003. co-located with ICSE'03.

[26] M. Lanza and S. Ducasse. A categorization of classes based on the visualization of their internal structure: the class blueprint. In *Proc. of the 16th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'01)*, pages 300–311, Tampa Bay, Florida, Oct. 14–18 2001. ACM SIGPLAN Notices.

[27] A. Lauder and S. Kent. Precise visual specification of design patterns. In *Proc. of the 12th European Conference on Object-Oriented Programming (ECOOP'98)*, Brussels, Belgium, July 20–24 1998. Springer.

[28] M. Lorenz and J. Kidd. *Object-Oriented Software Metrics: a practical guide*. Prentice-Hall, 1994.

[29] J. K. H. Mak, C. S. T. Choy, and D. P. K. Lun. Precise modeling of design patterns in UML. In *Proc. of the 26th International Conference on Software Engineering (ICSE'04)*, pages 252–261, Edinburgh, Scotland, United Kingdom, May 23-28 2004. IEEE Computer Society.

[30] B. Meyer. *Object-Oriented Software Construction*. International Series in Computer Science. Prentice-Hall, 1988.

[31] T. Mikkonen. Formalizing design patterns. In *Proc. of the 20th International Conference on Software Engineering (ICSE'98)*, pages 115–124, Kyoto, Japan, Apr. 19-25 1998. IEEE Comp.

[32] N. H. Minsky. Law-governed Linda communication model. Technical Report LCSR-TR-221, Dept. of Comp. Sc.Lab. for Comp. Sc. ResearchThe State Univ. of New Jersey RUTGERS, Mar. 1994.

[33] N. H. Minsky. Towards alias-free pointers. In *Proc. of the 10th European Conference on Object-Oriented Programming (ECOOP'96)*, pages 189–209, Linz, Austria, July 8–12 1996. Springer.

[34] J. Noble and R. Biddle. Patterns as signs. In *Proc. of the 16th European Conference on Object-Oriented Programming (ECOOP'02)*, Malaga, Spain, June 10–14 2002. Springer.

[35] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. An overview of the Scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[36] L. Prechelt and C. Krämer. Functionality versus practicality: Employing existing tools for recovering structural design patterns. *J.UCS: Journal of Universal Computer Science*, 4(12):866–882, 1998.

[37] M. Reiser and N. Wirth. *Programming in Oberon: steps beyond Pascal and Modula*. Addison-Wesley, June 1992.

[38] N. Schärli, S. Ducasse, O. Nierstrasz, and A. Black. Traits: Composable units of behavior. In *Proc. of the 17th European Conference on Object-Oriented Programming (ECOOP'03)*, Darmstadt, Germany, July 21–25 2003. Springer.

[39] J. M. Smith and D. Stotts. Elemental design patterns: A formal semantics for composition of OO software architecture. In *Proc. of the 27th Annual NASA Goddard Software Engineering Workshop (SEW'02)*, pages 183–190, Greenbelt, Maryland, Digital Equipment Corporation 5-6 2002. IEEE Comp. Soc. Press.

[40] E. Soloway, J. Bonar, and K. Ehrlich. Cognitive strategies and looping constructs: An empirical study. *Commun. ACM*, 26(11):853–860, 1983.

[41] J. T. Stasko. Tango: A framework and system for algorithm animation. *Computer*, 23(9):27–39, 1990.

[42] M. D. Storey and H. A. Müller. Manipulating and documenting software structures using SHriMP views. In *Proc. of the 11th International Conference on Software Maintenance (ICSM'95)*, page 275, Opio (Nice), France, Oct. 1995. IEEE Comp. Soc. Press.

[43] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3rd edition, 1997.

[44] S. T. Taft and R. A. Duff, editors. *Ada 95 Reference Manual, Language and Standard Libraries, International Standard ISO/IEC 8652: 1995(E)*, volume 1246 of *LNCS*. Springer, 1997.

[45] P. E. van Emde Boas. Resistance is futile; formal linguistic observations on design patterns. Technical Report ILLC-CT-1997-03, The Institute For Logic, Language, and Computation (ILLC), University of Amsterdam, Feb. 1997.

[46] N. Wirth. The programming language Pascal. *Acta Informatica*, 1:35–63, 1971.

[47] I. H. Witten and E. Frank. *Data mining: practical machine learning tools and techniques with Java implementations*. Morgan Kaufmann, 2000.