Implementation Patterns

Joseph (Yossi) Gil and Itay Maman

Department of Computer Science
Technion—Israel Institute of Technology

Abstract. An implementation pattern is a purposeful, simple and mechanically checkable predicate on the implementation of a software module. By definition, implementation patterns can be automated, thus supporting tasks such as discovery, validation, and even automatic application. They are also useful for recording knowledge, automatically augmenting documentation, validation, code inspection and style improvement. Moreover, implementation patterns provide a vocabulary which can be used not only between programmers, but also in describing implementation strategies of design patterns.

We present a catalog of 27 JAVA implementation patterns, formulated as predicates on classes and interfaces. Together, the patterns in the catalog capture 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 in object oriented programming.

Empirical validation included detection in data set of over 70,000 classes, in which we show that about 75% of these classes can be cataloged. Manual sample inspection confirms our belief that the cataloging process is sound. With very high statistical confidence level (often $\alpha < 0.01$) we can show that different libraries have different levels of pattern penetration, but that the relative ranking (Spearman correlation) of patterns tend to be similar. We also show that different implementations of the same specification make a significant tendency to use the same patterns. Further, the choice of an implementation pattern tends to be preserved in the course of class maintenance.

1 Introduction

A dozen years have passed since the publication of an ECOOP’93 paper entitled “Design Patterns: Abstraction and Reuse of Object-Oriented Design” [14]. This seminal work initiated a staggering research effort on the topic of design patterns, yielding thousands of articles and scores, if not hundreds of books. Still, attempts to automate and formalize design patterns are scarce. Systems like DisCo [18], LePUS [10, 11], the SPINE and HEDGEHOG systems [5], constraint diagrams [16], Elemental Design Patterns [23], 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., [7, 15]). Indeed, as Mak, Choy and Lun [17] say, “…automation support to the utilization of design patterns is still very limited.”

This paper present implementation patterns to complement the notion of design patterns. In a nutshell, implementation patterns are formal conditions on the attributes,
types, name and body of a software module and its components. An implementation pattern may carry an auxiliary set of recommendations, which are formal conditions similar to the main pattern condition, which can be used by automatic tools to generate tips to programmers on how to improve their code.

An implementation pattern captures a non-trivial idiom of the implementation language which serves a concrete purpose. The kinds of problems that implementation patterns solve are not as high level, nor context specific as those problems which design patterns solve.

Although implementation patterns are applicable to different languages and kinds of modules, we choose to concentrate on classes and interfaces of JAVA [4]. We present a catalog of 27 implementation patterns including idioms for a particular and intentionally restricted use of inheritance, immutability, wrapping, data management classes, OO emulation of procedural, modular and even functional programming paradigms, innovative use of class structure, and many more.

A simple example for concreteness is that of the Sampler pattern, that 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 reader is invited to take now a sneak preview of at the entire catalog in Sec. 4.)

Our empirical study demonstrates the consistent abundance of each of these idioms; further, we discover that this catalog characterizes about 75% of JAVA software.

1.1 Implementation Patterns vs. Design Patterns

The gap separating the initial imprecise and informal system requirement from the precise and formal manifestation of software in code written in a specific programming language cannot be crossed in a single leap. 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, an implementation pattern is not a strategy of implementation of a design pattern. For example, we discover that the Compound Box implementation pattern which is quite popular, is not acknowledged as a design pattern. There are however several obvious relations. For example, the Function Object implementation pattern, is very useful for implementing the COMMAND design pattern; Sampler is one implementation of the FLYWEIGHT design 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 Karmen [12]). Still, there are several consequences to the fact that implementation patterns stand at a lower level of abstraction.

First, implementation patterns are of a single software module in a particular programming language. The catalog presented for concreteness in Sec. 4 is all about individual JAVA classes and interfaces. 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.
Second, a crucial property of implementation patterns is that they are easily recognizable by software, which renders a smooth path to automation. Florijn, Meirjer and Winsen [13] enumerate three key issues in automating design patterns: application, validation and discovery. As van Emde Boas argues [26], 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. We give empirical evidence that implementation patterns can be discovered automatically. Their formal nature and associated formal recommendations help in semi-automatic validation and code improvement.

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 by Demsky and Rinard [9], 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.

Note that it is possible to deterministically check whether one implementation pattern is mutually exclusive, contained, or overlapping with another. In contrast, distinct design patterns, presented as solutions to two different problems, may have a “similar” structure (as is the case with STRATEGY and ADAPTER). There is therefore an inherent ambiguity in design patterns.

Third is the fact that implementation patterns cannot and do not 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. (Not surprisingly, much of the work on the automation of design patterns conveniently forgets the problem and the context.) An implementation 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.

A fourth difference, resulting from the loss of the problem and context in implementation 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, implementation patterns are not as detailed; 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 an implementation pattern will be found, but, admittedly, the nature of implementation patterns is such that they do not provide as much guidance as design patterns. On the other hand, the guidance that an implementation pattern does provide is suited for automation, and does not rely as much on abilities of the individual taking that guidance.

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.

1.2 Benefits of Implementation Patterns

Implementation patterns share many of the benefits of design patterns: They establish a vocabulary (for defining the different sorts of classes that can occur in large software systems). In fact, this vocabulary can come handy in the description of implementation strategies of design patterns. Terms such as Immutable, Box, Immutable Box, Pure Type or Implementor are useful in describing the implementation of patterns such as DECORATOR, BRIDGE, PROXY, etc. Also, architecture description languages such as ArchJava [2] may choose to use implementation patterns as part of their underlying constructs and vocabulary.

Just like design patterns, implementation patterns provide means for organizing and conveying a knowledge base. They are also useful for documentation, and help reduce learning time. And, they are applicable for reorganization and even refactoring (see e.g., Sec. 3) of existing software.

Beyond that, the fact that the catalog of implementation patterns is such a universal collection of provably useful programming techniques, makes it useful for a systematic study of the the culture of engineering software in the object oriented fashion.

The analysis information that our tools produce may enrich automatically generated documentation such as JavaDoc. This is especially important in the wake of the write once, use anywhere slogan which has brought about an explosion in the information that a programmer has to digest. For example, release 1.5 of JAVA spans circa 12,000 classes totalling some 84,000 methods. If each of these methods takes two minutes to study, then it would take a programmer well over a year to go over this huge collection.

A code inspector can choose to concentrate on the classes which do not fit into any of the patterns in the catalog, but also check the appropriateness of the assignment of classes.

Finally, the precise definition makes it possible to evolve some of the more important and common implementation patterns into language constructs. (The motivation for JAVA’s new enum facility is reflected by the prevalence of Augmented Type and Pool implementation pattern.) Agerbo and Cornils [1] discuss the incorporation of design patterns into a programming language.

Even though we believe that implementation patterns are more suited candidates to grow into language mechanisms, it is clear that this will not be the final destiny of most of the implementation patterns. First, the fact that implementation patterns are not as strict as compiler-enforced mechanisms makes the journey from the imprecise to the precise a little easier. Second, language mechanisms are slow to evolve and require deep pondering and careful evaluation of the expected benefits. The interaction of each such mechanism with all others must be considered-with hundreds of possible interactions in a catalog of patterns of our size.

Third, programmers should be allowed to add domain specific and in house patterns, and even modify the definitions of patterns in the catalog.

\[^1\] http://java.sun.com/j2se/javadoc
There have been a number of systems which allowed the programmer to add auxil-
iary, automatically checkable rules to code. Example include Minsky’s Law Governed
Regularities [19, 20], or Aldrich, Kostadinov and Chambers’s work on alias annota-
tion [3]. implementation 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.

1.3 Contribution

Beyond the introduction of this new notion, the main contributions of this paper are:

1. A catalog of implementation patterns, broken into eight categories, which can be
used not only for cataloging purposes but for introducing novice programmers to
the tools of the trade of JAVA and object oriented programming.
2. A format for describing patterns, which in trying to stick to Horn clauses, makes
automatic check and correction more convenient.
3. A concrete definition of the separation power of a catalog, also called entropy, as
well as the marginal entropy of each pattern in the catalog.
4. An empirical measurement of the coverage of catalog and its entropy in a large data
set.
5. A high confidence proof (in the statistical sense) that the variations in pattern preva-
ence levels in different software collections are meaningful. This proof supports
the claim that the implementation patterns describe a purposeful and meaningful
programming construct.
6. Another high confidence proof to support this claim, is that there is a correlation
between pattern prevalence levels and incidence in collections implementing simi-
lar specifications, and that implementation patterns tend to be preserved along the
life cycle of the software.

Outline. The remainder of this article is organized as follows. A definition of implemen-
tation patterns, together with an overview of key properties of these, make the subject
of Sec. 2. In Sec. 3 we show a template for a thorough description of an implementa-
tion pattern. Then, the complete catalog of implementation patterns is presented in Sec. 4.
Our data set of 15 large JAVA collections is presented in Sec. 5. Sec. 6 presents our
experimental results. Sec. 7 concludes.

2 Definition of Implementation Patterns and Catalog

Consider the following simple C++ function

```cpp
static inline int round(double d) {
    return d + 0.5;
}
```

We can distinguish four different and largely independent dimensions in the defi-
nition of this small module: (i) Body—which is as simple as return d + 0.5; here;
(ii) **Name**—identifier count, i.e., the function name; (iii) **Type**, i.e., the function signature, \( \text{int} \ (\ast) (\text{double}) \) in this case; and, (iv) **Attributes**—these are the **static** and **inline** keywords associated with the function.

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.

We wish to step further beyond the simple conclusion that there are many intercorrelations between the setting of (say) attributes and selection of types. We claim the distinct **patterns of implementation** which the majority of software follows, give meaning, name and significance to many of these correlations.

People use patterns without thinking. This use is a result of the recognition built into each one of us, that routine is easier and safer than the time consuming and error-prone process of decision making. Naturally, programmers will preserve energy to the more demanding tasks by using patterns as a mental skeleton to mold mundane modules.

3 Much of the coding work is therefore in selecting an implementation pattern for a module (often dictated by the system design), and then laboriously filling in many of the missing details to the unit in accordance with its implementation pattern.

One of the chief premises of the main claim is made by a mechanical analysis of a large software base, comprising close to a hundred thousand classes. Indeed, we find that an overwhelming majority of these classes follow one or more of the patterns in our catalog. The remaining classes either fit yet unknown patterns, or represent code locations which required more skill than routine.

An implementation pattern can be characterized as a **condition on the attributes, types, name and body of a module and its components**. However, not every such condition makes an implementation pattern. In order for a condition to be an implementation pattern, it must be **recognizable** (mechanically), **purposeful**, **prevalent** and **simple**.

Let us now discuss these four fundamental properties.

**Recognizability.** In saying “mechanically recognizable”, we mean that there exists a Turing machine which decides whether any given module matches this condition. Thus, the condition “the module delegates its responsibilities to others” is not recognizable. On 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 an implementation 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, implementation pattern **Immutable Box** describes classes which have a single instance field that is assigned only once, at construction time. This pattern

---

2 e.g., there are at least 42 different kinds of class members: 5 categories of class members (instance/static methods/fields, constructors), 4 visibility levels, and many modifiers such as **abstract** and **final**, etc.

3 In this paper, we focus attention to **JAVA** classes and interfaces, but the notion of implementation patterns should be applicable to smaller units, such as procedures and functions, just as to larger units, such as packages.
idiom typically serves the purpose of managing a single resource by a dedicated object, a practice which Stroustrup [24] 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 an implementation pattern is purposeful prevalent\(^4\). We denote the prevalence of a pattern \(p\) by \(\xi(p)\).

It is not possible to establish lower or upper thresholds for prevalence. The programming technique captured by Designator is so unique that a prevalence of 0.2\% is acceptable. The prevalence of Implementor, on the other hand, is circa 30\%.

We would like patterns to distinguish unique properties. Therefore, the observation \(\xi(p) > 0.5\) is a good sign that implementation pattern \(\neg p\) should be used instead of \(p\).

**Simplicity.** The simplicity requirement is not just a matter of aesthetics. An important application of implementation patterns is in CASE tools into which programmers enter an implementation patterns specification of the code. Such tools will not just verify that each module matches its implementation pattern classification, but also indicate concrete remedies in case of mismatch. In order for these indications to be useful, meaningful, and easy to interpret, the implementation pattern definition must be simple.

Consider for example the condition that all methods of a class must be implementations of abstract inherited ones (the Implementor implementation pattern). Suppose that a certain class violates that condition. Then, it possible to make a simple and clear indication of points in the code that lead to such violations.

As a counter example, consider the following condition (best expressed in second-order predicate logic):

\[
\text{There exist two disjoint subsets of fields, and two disjoint subsets of methods, such that all fields in the first (respectively second) set are used by all methods in the first (second) set, and all methods in the first (second) set use all fields in the first (second) set.}
\]

The application of formal concept theory to class design and inspection [8] suggests conditions such as the above capture a useful programming practice. However, although the above condition is trivially recognizable, it is difficult to indicate specific remedies in case of violation.

The sake of simplicity is served by expressing the conditions in first order predicate logic, whenever possible restricted in Horn clauses form. Simplicity is also the reason that out of the four dimensions of implementation, we favor conditions on attributes. Conditions on types are limited to type comparison, and avoid detailed analysis of the type structure. Our catalog does not involve any conditions on names.

It is sometimes necessary to write conditions on the actual body of the module. For example, for the purpose of detecting setter- and getter- methods, in the Data Manager pattern. However, such conditions tend to require complex code analysis (such as a “points-to” analysis required to conclude that a method delegates all its parameters to another method), and were only used when absolutely necessary.

\(^4\) Clearly, it is not sufficient—classes in which the number of methods is prime are prevalent, but have no common purpose.
2.1 Recommendations

The definition of an implementation pattern can be augmented by what we call an implementation pattern recommendation, or for short a recommendation, which is a similar predicate or a list of weighted predicates on a module, which include the less essential properties of an implementation pattern.

The predicate or predicates in the recommendation are used for giving a programmer tips on how to hone the implementation of the class to better serve its purpose. The purpose of the recommendation must be identical to the purpose of the pattern and its prevalence should be high (with respect to the base pattern). It is especially important that the statement of the predicate is such that an automatic tool is able to make specific indications on correcting a module so that it matches a recommendation.

2.2 Patterns Catalog

The fact that implementation 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. Given implementation patterns \( p_1 \) and \( p_2 \) 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 \( p_1 \land p_2 \).

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

2.3 Entropy, or Separation Power

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 amount of information that a catalog carries. First recall the definition of the information theoretical entropy.

**Definition 1.** The entropy of a distribution \( \xi_1, \ldots, \xi_k \), \( \sum_{1 \leq i \leq k} \xi_i = 1 \), and \( 0 \leq \xi_i \leq 1 \) is

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

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. 1, let us apply it to a single implementation pattern with prevalence \( \xi \) (with respect to some software collection). The entropy in this case is

\[
-\xi \log_2 \xi - (1 - \xi) \log_2(1 - \xi).
\]
Suppose that $\xi = \frac{1}{2^n}$. Then, the first summand states that the fact that implementation 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 implementation pattern does not occur. The entropy is 1 when the prevalence is 50%, it 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 2.** 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 $\xi(Q) = \xi(\bigwedge_{p \in Q} p)$.

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.

In order to evaluate the contribution of each implementation pattern to the expressive power of the catalog, we can examine its marginal contribution the catalog.

**Definition 3.** The marginal entropy of an implementation pattern $p \in P$ with respect to a catalog $P$, written $H(p/P)$ is $H(P) - H(P \setminus \{p\})$.

### 3 Implementation Patterns Presentation Format

Beyond the actual definition of the condition in the implementation pattern, there is much information to be conveyed to a programmer in order to make it useful. This section presents a template for a detailed description of an implementation pattern. Pattern Overrider is used as a concrete running example.

There are five chapters in this description: **Identification, Properties, Applicability, Code Sample and Consequences**.

The first and most important chapter is **Identification**, which defines and classifies the pattern, including the following items:

**Name and Nick Names (Abbreviation)** Overrider (Ovrd)

This is the pattern name along with a four letter abridged version. Short, descriptive, meaningful and memorable names are essential for an effective vocabulary.

**Categories** Inheritors

This is the category or categories into which the pattern belongs. The list of categories appears in the next section.

**Purpose** A class that modifies, but does not add to, the implementation of its parent.

This item briefly describes the primary purpose that this pattern serves.
Definition

\[ p(C) = \forall m \in C \cdot m \in \text{Methods} \land m \notin \text{Constructor} \cup \text{private} \cup \text{static} \implies m \notin \text{abstract} \land m \in \text{parent}(C) \land m \notin \text{parent}(C).\text{abstract} \]

This is a definition of the condition of the implementation pattern. It can be formulated in natural language, provided that its translation to first order predicate calculus is straightforward. The preferred format is that of Horn clauses. The above definition states that all methods of a class which are not constructors, private or static must appear in the parent of the class, and be non-abstract there. (Note the notation in which keywords denotes sets, e.g., private is the set of all private class members.)

Recommendation

1. \( q_1(C) = \forall m \in C \cdot m \notin \text{static} \).
2. \( q_2(C) = \forall m \in C \cdot m \in \text{Methods} \implies \text{calls}(m, \text{parent}(c).m) \)

This is a statement of non-essential requirements from the pattern instance. In this case, it is the recommendation that the class does not define any static elements, and (with lower priority) that the methods that \( C \) defines refine those of its parent.

Example java.io.BufferedOutputStream

A typical class matching the implementation pattern, taken from the standard JRE library.

The second chapter is Properties, which gives statistical and other properties of the pattern. This chapter includes the following items:

Prevalence 11% ± 7%

These are supplied from actual measurement in our data set, where the standard deviation is taken across different collections.

Marginal Entropy 0.23

Computed with respect to the catalog and the entire data set

Correlation Sink, Immutable Box

Specifies which other implementation patterns have significant positive correlation association with the described pattern, i.e., tend to appear together in the same class.

Excludes Designator, Augmented Type, Implementor, Joiner, Monostate, Pseudo Class, Pool, Record, Taxonomy

Lists the implementation patterns that can never occur together with the described pattern. (There is no “negative correlation” item, since usually pattern prevalence is not high enough to make significant negative correlation values.)

The Applicability chapter describes typical scenarios where the implementation pattern is used. For example,

The Overrider pattern is used in cases where the desired functionality of a new class \( C \) is similar to that of an existing class \( B \). In particular, the changes to \( B \) are to limited to modifications of its behavior but not of its protocol. Many times, the methods of \( C \) will refine the corresponding methods of \( B \).
There is also a **Code Sample** chapter, which is a short JAVA source code illustrating the implementation pattern. For example,

```java
public class MyWindowAdapter extends WindowAdapter {
    public void windowClosing(WindowEvent e) {
        System.exit(0);
    }
}
```

Finally, the **Consequences** chapter of the implementation pattern description provides additional characteristic which are frequently exhibited by the class, and advices to the user. For example,

> Often the modifications that $C$ performs on $B$ are not a coincidence, but rather represent a deeper notion. Search for other classes which make modifications to $B$ or to $C$. These will tend to be instances of Overrider as well. Consider refactoring the common parts of classes $B$ and $C$ into a new class from which both classes will inherit.

## 4 A Catalog of Java Implementation Patterns

This section enumerates the 27 implementation patterns in our catalog, describing each one of them briefly (sometimes extremely briefly). The full format described in the previous section could not be used due to space limitations.

Classes can be sub-classed, and they can be implemented by inheriting other classes. We therefore have a category of implementation patterns that describe classes intended to be used as basis for inheritance, and a counter category of patterns describing inheritor classes. Classes are also characterized by state (fields) and a by behavior (methods). There are categories of patterns if one or both of these properties is degenerate.

Eight categories of implementation patterns are described in the subsequent eight subsections. The categories are not disjoint. For example, the Pseudo Class pattern belongs both to the **Degenerate State and Behavior** and the **Base Classes** categories. Such patterns are described in only one of their categories, and mentioned in the others.

### 4.1 Degenerate State and Behavior

The first, and most simple category of implementation 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 implementation patterns which are described in the **Base Classes** category.

1. **Designator.** The most trivial interface is an empty one. The Designator implementation pattern captures those interfaces which do not declare any methods, do not define any
static fields or methods, and do not inherit any such declarations or interfaces. Interfaces of this sort demonstrate 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 Cloneable\(^5\) indicates (at run time) that it is legal to make a field-for-field copy of instances of that class. Although a rarity, a class can also be Designator if its definition, as well as the definitions of all of its ancestors (other than Object), are empty.

2. Taxonomy. Even if the definition of an interface is empty it may still extend another, potentially non-empty, interface. 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. An example is interface DocAttribute in package javax.print.attribute, which extends interface Attribute in the same package without adding any further declarations. Similarly to the Designator implementation pattern, this interface is used for tagging purposes—specifically that the attribute at hand is specialized for what is known as “Doc” in the JRE.

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 implementation 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 thrown exception’s type, 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 different interfaces: interface MouseListener and interface MouseMotionListener.

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 has no instance fields. Moreover, all of its static fields must be final.\(^6\) Another requirement is that the class has no methods (other than those inherited from Object, or automatically generated constructors).

Programmers often use interfaces for the Pool implementation pattern. For example, the interface javax.swing.SwingConstants defines constants used in positioning and orienting screen components. This “constant interface anti-pattern” \(^[6]\) makes it possible to incorporate a name space of definitions into a class by adding an implements clause to that class.

---

\(^5\) This, and all subsequent examples are drawn from the standard Java Runtime Library.

\(^6\) 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.
4.2 Degenerate Behavior

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

5. Record. JAVA makes it possible to define classes which look and feel much like PASCAL record types. A class matches the Record implementation pattern if all of its fields are public and has no methods other than constructors and methods inherited from Object. Perhaps surprisingly, there is a number of examples of this pattern in the JAVA standard library. For example, java.sql.DriverManager is a record managing a textual property passed to a JDBC driver.

6. Data Manager. Experienced object-oriented programmers will encapsulate all fields 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.

7. Function Pointer. Very peculiar are those classes which have no fields at all, and only a single public instance method. Thus allowing an object to represent function or a procedure in the procedural programming paradigm. For instance, the class com.sun.jndi.ldap.LdapNameParser has a single parse method, with (as expected) a string parameter.

8. Function Object. The Function Object implementation pattern is similar to the Function Pointer implementation 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 of the 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, these classes often realize the COMMAND pattern.

9. Cobol like. Formally, a Cobol like 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 (better called micro pattern) is vanishingly small, instances can be found even in mature libraries. Note that beginner programmers may tend to use Cobol like for their main class, i.e., the class with function

    public static void main(String[] args)

4.3 Degenerate State

This category includes classes whose instances have no state at all, or their state is shared by other classes, or that they are immutable. This category also contains the Mould pattern which is defined under the Base Classes category.

10. Stateless. If a class has no fields at all (except for static 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. 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 Stateless implementation pattern is the java.util.Arrays class.

11. Monostate. At the next level of complexity, stand classes that maintain state, but this state is shared by all of their instances. Specifically, such a class has no instance fields, but at least one static field. For example, the class System manages the global input, output, and error streams. implementation pattern is in fact an incarnation of the modular\(^8\) programming paradigm in the JAVA world.

12. Immutable Box. A class with exactly one instance field which is only changed by its constructors. Integer is such a class.

13. Immutable. A class whose instance fields are only changed by its constructors. Must have at least two instance fields. java.util.jar.Manifest is an Immutable class since assignment to its two fields takes place only in constructors code.

4.4 Wrappers

Wrappers are classes which wrap a central instance field with other methods. They tend to delegate operation to this field.

14. Box. A 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 implementation pattern. Its entire state is represented by a single field (int crc), which is mutated by method update(int i).

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.

4.5 Data Managers

Data managers are classes whose main purpose is to manage the data stored in a set of instance variables. Included here are Record, DataManager as well as,

16. Sink. A class where its declared methods do not call neither instance methods nor static methods. java.util.jar.JarEntry is a Sink class.

4.6 Inheritors

The three disjoint patterns in this category correspond to three different ways in which a class can use the definitions in its super class: 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.

\(^8\) The term “modular” means here module-oriented, where modules are software units such as ADA [25] packages and modules in MODULA [22].
17. Implementor. An Implementor is a non-abstract class such that all of its the public methods were declared as abstract in its super class. For example, class SimpleFormatter, defined in package java.util.logging, has a single public method, format(LogRecord logrecord), which was declared abstract by the super class, Formatter (of the same package).

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

19. Extender. An Extender is a class which extends the interface inherited from its super class and super interfaces, but does not override any method. For example, class Properties (in java.util) extends its super class (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 super class behavior intact. Note that an Extender may be regarded as an instantiation of a degenerate mixin class over its super class.

4.7 Base Classes

This category includes specific ways in which a class can make preparations for its subclasses.

20. Outline. An abstract class where two or more declared method invoke at least one abstract methods of the current (“this”) object. 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.

21. Mould. The Mould patterns captures abstract classes which have no state. Specifically, a Mould class must have no instance fields, and at least one abstract method. For instance, class Number (from java.lang) provides an implementation for method shortValue() and byteValue(), but other than that, it expects its sub class to provide both implementation and state.

22. State Machine. It is not uncommon for an interface to define only parameterless methods. This type of 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 can be 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. The java.util.Iterator interface, for example, 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.

23. Pure Type. A class that lacks all implementation details is a Pure Type. Specifically, the requirements are that the class is abstract, has no static members, it has 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.
24. 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 \(^9\). 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. Therefore, `java.sql.Statement` 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.

25. 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.

### 4.8 Controlled Creation

Patterns in this category are concerned with the issue of constructing their instances. The first pattern in the category prevents clients from creating instances directly. The second pattern in the category provides to clients ready made instances.

26. Restricted Creation. A class with no public constructors, and at least one `static` field of the same type as the class. Many SINGLETON classes satisfy this criteria, such as `java.lang.Runtime`.

27. Sampler. A 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. See `java.awt.Color`.

### 5 Data Set

An ensemble of fifteen 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. Table 1 summarizes some of the essential size parameters of these collections. The table does not include a line count of the collections in the data set, 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

---

\(^9\) The reason is that a set of size two is in many cases best represented as boolean.
<table>
<thead>
<tr>
<th>Collection</th>
<th>Packages</th>
<th>Classes</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kaffe$^{1.1}$</td>
<td>75</td>
<td>1,220</td>
<td>10,945</td>
</tr>
<tr>
<td>Kaffe$^{1.1.4}$</td>
<td>152</td>
<td>2,511</td>
<td>22,022</td>
</tr>
<tr>
<td>Sun$^{1.1}$</td>
<td>67</td>
<td>991</td>
<td>9,448</td>
</tr>
<tr>
<td>Sun$^{1.2}$</td>
<td>131</td>
<td>4,336</td>
<td>36,661</td>
</tr>
<tr>
<td>Sun$^{1.3}$</td>
<td>170</td>
<td>5,213</td>
<td>44,747</td>
</tr>
<tr>
<td>Sun$^{1.4.1}$</td>
<td>314</td>
<td>8,216</td>
<td>73,834</td>
</tr>
<tr>
<td>Sun$^{1.4.2}$</td>
<td>330</td>
<td>8,740</td>
<td>76,675</td>
</tr>
<tr>
<td>Scala</td>
<td>96</td>
<td>3,382</td>
<td>32,008</td>
</tr>
<tr>
<td>MJC</td>
<td>41</td>
<td>1,141</td>
<td>10,927</td>
</tr>
<tr>
<td>Ant</td>
<td>120</td>
<td>1,970</td>
<td>17,911</td>
</tr>
<tr>
<td>JEdit</td>
<td>23</td>
<td>805</td>
<td>6,110</td>
</tr>
<tr>
<td>Tomcat</td>
<td>280</td>
<td>4,335</td>
<td>43,877</td>
</tr>
<tr>
<td>Poseidon</td>
<td>594</td>
<td>10,052</td>
<td>77,988</td>
</tr>
<tr>
<td>JBoss</td>
<td>998</td>
<td>18,699</td>
<td>157,460</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>3,391</strong></td>
<td><strong>71,611</strong></td>
<td><strong>620,613</strong></td>
</tr>
</tbody>
</table>

Table 1. The JAVA class collections comprising the data set

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 ensemble 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 Table 1.

Second, to compare the incidence of implementation 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\(^\text{10}\)—which is a non-commercial JVM implementation. Our ensemble 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 data set 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 data set 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 ensemble includes two GUI applications: JEdit—which is version 4.2 of the programmer’s text editor written in JAVA with a Swing GUI,

\(^{10}\) http://www.kaffe.org
and Poseidon—a popular UML modeling tool delivered by Gentleware. (We used version 2.5.1 of the community edition of the product.)

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

4. **Compilers and Language Tools.** These include: Ant—another component of the Apache project—a build tool which offers functionality that is similar, in principle, to the popular make utility (on version 1.6.2); Scala—version 1.3.0.4 of the implementation of the Scala multi-paradigm programming language; 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 ensemble 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).

<table>
<thead>
<tr>
<th>Collection</th>
<th>Packages</th>
<th>Classes</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sun 1.4.2</td>
<td>272</td>
<td>7,525</td>
<td>66,676</td>
</tr>
<tr>
<td>Scala</td>
<td>68</td>
<td>2,678</td>
<td>25,186</td>
</tr>
<tr>
<td>MJC</td>
<td>32</td>
<td>945</td>
<td>8,607</td>
</tr>
<tr>
<td>Ant</td>
<td>45</td>
<td>421</td>
<td>3,883</td>
</tr>
<tr>
<td>JEdit</td>
<td>21</td>
<td>676</td>
<td>4,653</td>
</tr>
<tr>
<td>Tomcat</td>
<td>132</td>
<td>1,434</td>
<td>14,367</td>
</tr>
<tr>
<td>Poseidon</td>
<td>477</td>
<td>8,162</td>
<td>61,645</td>
</tr>
<tr>
<td>JBoss</td>
<td>750</td>
<td>13,623</td>
<td>110,820</td>
</tr>
<tr>
<td>Shared</td>
<td>346</td>
<td>5,979</td>
<td>55,306</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>2,143</strong></td>
<td><strong>41,443</strong></td>
<td><strong>351,143</strong></td>
</tr>
</tbody>
</table>

Table 2. The JAVA class collections in the pruned data set

Note however that the totals in the last line of Table 1 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 authors 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. The
size of the nine collections in the pruned ensemble are reported in Table 2. We can see that the elimination of duplicates and dependent implementations halved the size of the ensemble. In total, more than 41,000 independent class comprise the pruned ensemble.

6 Experimental Results

Table 3 summarizes the bulk of our experimental findings, supplying prevalence level, and marginal entropy for each pattern and coverage and entropy of the entire catalog with respect to each of the collections in the pruned data set.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Taxonomy</td>
<td>0.94</td>
<td>0.93</td>
<td>0.88</td>
<td>0.96</td>
<td>0.90</td>
<td>0.89</td>
<td>0.93</td>
</tr>
<tr>
<td>Joiner</td>
<td>0.74</td>
<td>0.42</td>
<td>0.06</td>
<td>0.96</td>
<td>0.92</td>
<td>0.89</td>
<td>0.91</td>
</tr>
<tr>
<td>Leaf</td>
<td>1.00</td>
<td>0.49</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
</tbody>
</table>

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 Sun142 are cataloged. The collection with least coverage is Art, but even for it, one in two classes is cataloged. The total coverage of the (pruned) data set is 74.9%. The fluctuation in coverage level is not very great—the standard deviation (penultimate column) is 11%.

Conclusion 1 Three out of four classes match at least one implementation pattern in the catalog.

As explained above, many of the implementation patterns are not mutually exclusive. There are classes in the data set which match more than one implementation pattern. We found that in the pruned data set, 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 data set there was also a significant number of 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!

Obviously, a multi-pattern attribution of a class carries more information than a single pattern label. The entropy of the catalog tells us how much information (in the information theoretic sense) the catalog provides on average. Examining the last row of the table, we learn that the entropy fluctuates between 4.28 and 5.27. By raising these values to the power of two, we obtain,

**Conclusion 2** *The separation power of the catalog is equivalent to that of a partitioning into 19–39 equal and disjoint sets.*

The last column of Table 3 gives the marginal entropy of each of the implementation pattern with respect to the entire catalog and the pruned data set. 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 implementation pattern.

If a pattern is identical to another pattern in the catalog, or to any combination of these, then its marginal entropy is 0. 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.

We obtain,

**Conclusion 3** *All patterns contribute to the separation power of the catalog.*

In examining this column we also find that patterns with high prevalence tend to have 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.

The body of Table 3 presents, for each implementation pattern and each software collection, the *prevalence* of the implementation pattern in the collection, that is, the percentage of classes in this collection which match this implementation pattern. (Note that due to overlap between the patterns, columns do not add up to the total coverage in the last row.)

Following this 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) data set, 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 implementation patterns, these three typical values are close to each other. (This situation is typical to symmetrical 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 see,

**Conclusion 4** *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.

We see Conc. 4 as a supporting evidence to our claim that the patterns in the catalog are purposeful. We expect different collections to have different needs, and therefore employ different patterns at different levels. Alternatively, these differences could reflect difference in style of the design team.

The null hypothesis $H_0$ is that patterns are a random property of JAVA code, independent of context and programming style. In other words, that the changes in prevalence across different collections is due to normal fluctuations of random values. In attempt to reject this null hypothesis, we applied a standard Chi-squared tests to all pairs of columns in Table 3. This test checks whether a random fluctuations in prevalence values can explain with any reasonable probability the differences between two columns, i.e., two collections.

We find that all of the 36 pairs of collections (represented as columns in the table), passed the test with confidence level of 99.9%, i.e., $\alpha < 0.001$. Thus, hypothesis $H_0(c_1, c_2)$ is rejected for all pairs $\langle c_1, c_2 \rangle$ of collections. In fact, it is rejected with the same confidence level for all 225 pairs of collections in the full data base, including pairs of two consecutive implementations of the JRE. We can therefore argue,

**Conclusion 5** The catalog of implementation patterns is effective for an automatic detection of changes in programming practice between software collections.

Note that Conc. 5 does not give us means of understanding the changes. It merely says that these changes as a whole are significant.

Moreover, the rejection of $H_0$ does not mean that every change in prevalence level of each pattern is significant; despite the great variety, some patterns show at the same prevalence level in different collections. We can check however for each individual implementation pattern $p$, the null hypothesis $H_0(p)$ that states in prevalence level in different collections are merely a matter of random fluctuation. Hypothesis $H_0(p)$ is rejected at the 99.9% confidence level for all $p$. Therefore,

**Conclusion 6** Changes in prevalence of an individual implementation pattern in the collections of the data set are significant.

Again, the above does not mean that every change in every prevalence level is significant, but rather that not all changes in the collections are a matter of coincidence.

In examining Table 3 in greater detail, we see that the most prevalent group is this of the inheritors implementation 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 implementation pattern, in this category and overall, is Implementor which occurs in about 21% of all classes. 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%. 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 data set.

**Conclusion 7** The majority of classes are cataloged by one of the five leading patterns.

The rarest pattern is Designator. (It was included in the catalog because it presents an important Java technique, which is also easily discernible.) Two “anti-patterns” in the catalog, Pseudo Class and Cobol like are also rarities (0.4%, 0.5%, 0.5% prevalence, respectively.) The Augmented Type pattern is also rare (0.5%), perhaps thanks to the advent of the Enum mechanism to the language.

We argue that a consistent prevalence, even of 0.5%, in such a huge data set is significant.

<table>
<thead>
<tr>
<th>Collection</th>
<th>Avg %</th>
<th>Std Dev %</th>
<th>Min %</th>
<th>Max %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Designator</td>
<td>1.3%</td>
<td>0.8%</td>
<td>0.7%</td>
<td>1.9%</td>
</tr>
<tr>
<td>Taxonomy</td>
<td>11.0%</td>
<td>6.7%</td>
<td>5.8%</td>
<td>16.8%</td>
</tr>
<tr>
<td>J2EE</td>
<td>1.9%</td>
<td>0.9%</td>
<td>1.2%</td>
<td>2.7%</td>
</tr>
<tr>
<td>J2SE</td>
<td>0.4%</td>
<td>0.3%</td>
<td>0.8%</td>
<td>1.2%</td>
</tr>
<tr>
<td>J2CL</td>
<td>1.9%</td>
<td>1.4%</td>
<td>1.6%</td>
<td>2.3%</td>
</tr>
<tr>
<td>J2SE</td>
<td>0.2%</td>
<td>0.2%</td>
<td>0.0%</td>
<td>0.5%</td>
</tr>
<tr>
<td>Designator</td>
<td>2.7%</td>
<td>2.8%</td>
<td>1.5%</td>
<td>3.9%</td>
</tr>
<tr>
<td>Taxonomy</td>
<td>1.0%</td>
<td>1.0%</td>
<td>2.1%</td>
<td>2.1%</td>
</tr>
<tr>
<td>J2EE</td>
<td>1.1%</td>
<td>1.2%</td>
<td>1.8%</td>
<td>6.0%</td>
</tr>
<tr>
<td>J2SE</td>
<td>0.9%</td>
<td>0.6%</td>
<td>0.4%</td>
<td>0.4%</td>
</tr>
<tr>
<td>J2CL</td>
<td>1.0%</td>
<td>1.0%</td>
<td>2.1%</td>
<td>2.9%</td>
</tr>
<tr>
<td>Stateless</td>
<td>5.0%</td>
<td>7.9%</td>
<td>9.3%</td>
<td>12.3%</td>
</tr>
<tr>
<td>Monostate</td>
<td>2.0%</td>
<td>1.9%</td>
<td>2.8%</td>
<td>4.9%</td>
</tr>
<tr>
<td>Immutable Box</td>
<td>1.0%</td>
<td>1.0%</td>
<td>2.1%</td>
<td>2.1%</td>
</tr>
<tr>
<td>Immutable Box</td>
<td>1.0%</td>
<td>1.0%</td>
<td>2.1%</td>
<td>2.1%</td>
</tr>
<tr>
<td>Box</td>
<td>8.0%</td>
<td>8.0%</td>
<td>9.0%</td>
<td>10.0%</td>
</tr>
<tr>
<td>Compound Box</td>
<td>7.6%</td>
<td>6.6%</td>
<td>8.4%</td>
<td>4.4%</td>
</tr>
<tr>
<td>Implementor</td>
<td>9.7%</td>
<td>9.7%</td>
<td>9.7%</td>
<td>9.7%</td>
</tr>
<tr>
<td>Overrider</td>
<td>6.6%</td>
<td>5.5%</td>
<td>7.7%</td>
<td>9.2%</td>
</tr>
<tr>
<td>Extender</td>
<td>6.0%</td>
<td>4.3%</td>
<td>4.1%</td>
<td>4.3%</td>
</tr>
<tr>
<td>Orifice</td>
<td>1.0%</td>
<td>2.8%</td>
<td>2.3%</td>
<td>3.7%</td>
</tr>
<tr>
<td>Mould</td>
<td>1.0%</td>
<td>1.0%</td>
<td>1.0%</td>
<td>1.0%</td>
</tr>
<tr>
<td>State Machine</td>
<td>1.0%</td>
<td>1.0%</td>
<td>1.0%</td>
<td>1.0%</td>
</tr>
<tr>
<td>Pure Type</td>
<td>1.0%</td>
<td>1.0%</td>
<td>1.0%</td>
<td>1.0%</td>
</tr>
<tr>
<td>Augmented Type</td>
<td>1.0%</td>
<td>1.0%</td>
<td>1.0%</td>
<td>1.0%</td>
</tr>
<tr>
<td>Pseudo Class</td>
<td>0.9%</td>
<td>0.8%</td>
<td>0.7%</td>
<td>0.9%</td>
</tr>
<tr>
<td>Sink</td>
<td>21.9%</td>
<td>27.5%</td>
<td>17.0%</td>
<td>17.0%</td>
</tr>
<tr>
<td>Restricted Creation</td>
<td>2.0%</td>
<td>2.5%</td>
<td>1.2%</td>
<td>1.6%</td>
</tr>
<tr>
<td>Entropy</td>
<td>4.9%</td>
<td>5.0%</td>
<td>4.6%</td>
<td>5.2%</td>
</tr>
</tbody>
</table>

Table 4. The prevalence, coverage and entropy of implementation patterns in different implementations of the JRE.
We now turn to the quest of checking the persistence of implementation 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. 5. Table 4 has a similar structure to that of Table 3 except that in Table 4 we compare the implementation pattern prevalence in the seven implementations of the JRE.

Comparing the two tables see that the values in the average, total, and median lines in Table 4 are close, just as they are in Table 3.

In comparing the standard deviation column ($\sigma$) in the two tables, we see that the variety in coverage level and entropy is much smaller in the related collections (Table 4) than the variety in the related collections (Table 3). For the majority of patterns (20 out of the 27), the variety in prevalence level in Table 4 is smaller than the variety in Table 4.

The variety of four patterns is about the same in both data sets. Only three patterns, Designator, Taxonomy, and Immutable, 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 most of the classes in it. The drop in Designator and Taxonomy 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 use of new classes like these.

**Conclusion 8** Pattern prevalence tends to be the same in software which serves similar purposes, independent of the size of the collection.

This conclusion may be a bit surprising since the different versions of the implementations of the JRE have a very different total number of classes.

It may even take moment’s thought to be convinced that the above Conc. 8 does not contradict Conc. 5. What we have in fact is that although the prevalence level tends to be similar in similar implementations, changes in the prevalence level (across all patterns) are statistically significant.

This claim can be made more precise in the following sense: we can show that there is a statistically significant correlation (with confidence level $\alpha < 0.05$), in each of the following: (i) the events that a class was implemented using a certain implementation pattern in one version of the JRE and in another version of the JRE by another vendor; (ii) the events that a class was implemented using a certain implementation pattern in one version of the JRE and a subsequent version of the JRE; (iii) the prevalence level of patterns, i.e., Pearson correlation, in different versions of the JRE; (iv) the ranking (by prevalence) of patterns in different versions of the JRE, i.e, the Spearman correlation.

Although the details are omitted, one might see why this holds by examining the difference in prevalence of a single implementation pattern in two consecutive version of Sun’s JRE, we see that in most of the cases, the difference is not larger than 1% (median value of all such differences is 1%).
7 Conclusions and Further Research

The bountiful class structure of JAVA, together with the colossal, publicly available, base of software in the language, opens the road to much exciting research of implementation patterns, in attempt to understand the way people write software. Sec. 6 employed techniques of statistics and social sciences research, often considered foreign to our field, to study the “large population of JAVA classes as written by Humans”.

We believe this is just a first step in a new direction: we can engage in research to check whether certain patterns tend to communicate with other patterns in specific ways. A typical research question of this sort is whether base class category is indeed used more as inheritance basis. Also interesting is the Implementor pattern: is it statistically true that it typically presents an opportunity for factorization?

Relying on the work of Demsky and Rinard [9], we argued that it is possible to build systems which will not only discover the use of implementation patterns, but also help the user correct his software accordingly. Experience in doing so would serve the community.

Another direction of research is to develop the mathematical foundation of pattern composition and decomposition, in at least two dimensions: First, it is interesting to know how should the pattern definition be made so that a class can fit more than one specification, thus enhancing the separation power of the dictionary. Second, there is room for research on the kinds of interactions between classes obeying various implementation patterns and even developing implementation patterns to specify sorts of such interaction.

References