The Use of Overloading in JAVA Programs

Joseph (Yossi) Gil[1]* and Keren Lenz[2]

[1] IBM Haifa Research Laboratory, Israel
[2] Department of Computer Science, The Technion, Haifa, Israel

Abstract. Method overloading is a controversial language feature, especially in the context of Object Oriented languages, where its interaction with overriding may lead to confusing semantics. One of the main arguments against overloading is that it can be abused by assigning the same identity to conceptually different methods.
This paper describes a study of the actual use of overloading in JAVA. To this end, we developed a taxonomy of classification of the use of overloading, and applied it to a large JAVA corpus comprising more than 100,000 user defined types.
We found that more than 14% of the methods in the corpus are overloaded. Using sampling and evaluation by human raters we found that about 60% of overloaded methods follow one of the "non ad hoc use of overloading patterns" and that additional 20% can be easily rewritten in this form. The most common pattern is the use of overloading as an emulation of default arguments, a mechanism which does not exist in JAVA.

1 Introduction

$208,765,973,875,851$, the count of distinct *admissible* identifiers in early versions of C [15], may seem a fairly large number. Still, as large as this number is, it is infinitesimally small when compared to its JAVA [1] counterpart. Yet, *adequate* identifier names are hard to come by, both in JAVA and in C, as anyone who tried naming a programming entity—be it a variable, a function, or a newly introduced type—must have noticed: the problem is not of finding the needle in the haystack, but the simple truth that, no matter how large the universe of discourse is, the competition on the few scarce good names remains fierce.

Striking a balance between the desire to make names descriptive and meaningful, and the practical demand that these are not overly verbose, we often wish to use identifiers such as `print`, `close`, `sort`, `execute` or `draw` in reference to distinct entities. Program blocks and scoping rules serve this wish in making it possible to reuse a name in *different* contexts in an orderly fashion. A common, yet controversial mechanism for reusing a name within the *same* context, is *overloading*, an ad-hoc kind of polymorphism [5].

Several style guides [3] all but completely forbid the use of overloading. This practice could be justified e.g., by the vigorous criticism by B. Meyer [18], expressed succinctly with his, almost axiomatically-true, statement:

* On sabbatical from the Technion
[3] http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml

Different Things Should have Different Names

But, this statement could be (and often is) answered by an equally self-evident truth

The Same Things Should Have the Same Name

which reveals the clumsiness in the encoding function signatures into their names, e.g., in the definition of a series of functions:

- `printInt(int i)`,
- `printBoolean(boolean b)`,
- `printChar(char c)`, etc.,

instead of straightforward use of overloading: `print(int i)`, `print(boolean b)`, `print(char c)`, etc.

Meyer and others [4] point a finger at the ambiguity innate in overloading—an ambiguity which is exacerbated in the presence of inheritance, genericity, coercion, and language-specific mechanisms (e.g., non-**`explicit`**, single parameter constructors in C++ [20], covariance in EIFFEL [14], etc.). Arguably, setting the rules for resolving this ambiguity may require a hefty load of language legalese, and a not so pleasant challenge to the unsuspecting programmer. Suffice to say that even the semantics of the trivial case of overriding one of two overloaded versions of a function is different in JAVA and in C++.

Constructors pinpoint the difference in opinion between the parties to this debate: JAVA, C++ and C# [13] programmers are not free to name constructors as they please—all constructors of a given class must bear its name. Since constructors are not inherited, at least the intricacies of interaction between overloading and inheritance are saved. Still, even supporters may see flaws in constructor overloading. To quote a JAVAWorld article:[4]

> *"With* JAVA*, the language design for constructors is quite elegant—so elegant, in fact, that it's tempting to provide a host of overloaded constructors. When the number of configuration parameters for a component is large, there can be a combinatorial explosion in constructors, ultimately leading to a malady known as* constructor madness. . . *"*

1.1 This Work

In this paper, we contribute to the discussion between proponents and opponents of overloading by a study of the use of overloading in JAVA programs. For this study, we developed a taxonomy of categories (which can also be called patterns and even *micro-patterns* [11]), for the classification of the use of overloading, based mostly on the type of interaction between overloaded methods. This taxonomy is also characterized by stretching a spectrum of the use of overloading, from ad hoc patterns, in which overloading is coincedental, to systematic patterns, in which overloaded methods are semantically cohesive.

[4] "Java Tip 63, Jerry Smith, Nov. 1, 1998"

In order to estimate the prevalence of the various overloading patterns in actual code we conducted an empirical evaluation, in which we applied this taxonomy to a large corpus of JAVA applications using a new research method. This method includes randomly sampling the corpus, manually evaluating the sampled items and testing the reliability of this evaluation, while employing techniques traditionally used in social sciences. Also of interest is the way in which the development of the taxonomy was in tandem with the two batches of work by human raters, and how the reliability of the human classification was estimated. This research method, to the best of our knowledge, was not previously applied to the study of software.

In the empirical evaluation we sought to answer the following questions:

1. What is the probability that a method, selected at random from the corpus, is overloaded?
2. What is the probability that a constructor, selected at random from the corpus, is overloaded?
3. For each of the overloading pattern, what is the probability that a method (or a constructor), selected at random, follows this pattern?

The answers to these questions provide evidence that overloading is used extensively in Java programs, and that, in contrast with the predictions of its opponents, overloading is used mostly in a systematic fashion.

The use of overloaded functions to implement a similar, but slightly different semantics, does prove that programmers do not abuse the mechanism. At the same time, even systematic use of overloading is not so desired from a software engineering standpoint. For the class's author, this means a blown up interface with extra code to document and maintain. For the class's client, this practice requires familiarity with different versions of essentially the same method.

Moreover, the semantics of the interaction between overloading and overriding varies between languages [4]. Understanding this subtlety is requiered in order to make sure that the intended method is indeed invoked. The example in Figure 1, drawn from [4], illustrates the problem.

Class `Down` presented in this figure overloads methods `f` and `g` introduced in its super class. Now, consider the following invocations:

```
(new Down()).f(new Top());
(new Down()).g(new Bottom());
```

Which methods get called? The answer depends on the language in which this model is implemented. In JAVA, both calls invoke `Up`'s methods, while in C++ the first call results in an error and the second invokes `Down`'s `g`. The reason for these differences is that in C++, `Down`'s methods *hide* those of `Up` rather than overload them.

1.2 On Empirical Study of Programming Languages

The design of an object oriented programming language, just as an extension of one, is an art in many ways. In other ways, it is an exact science, requiring rigorous analysis of semantics, soundness, etc. and of course, exciting engineering is also involved. But, do we really understand how this tool is really used, or abused?

Both issues of data gathering and data analysis are what makes it difficult to understand how the industry really uses a programming language. But, these difficulties should not stop us from trying.

This paper offers, in a sense, one direction at which such understanding may be gained. First, it uses Qualitas corpus[5], an organized collection of software systems intended to be used for empirical studies in software engineering. Observing the size and the increasing acceptance of this corpus we can say that we are getting closer to a meaningful sample of the global concrete use of JAVA.

The issue of data analysis remains. Exact static analysis techniques are prohibitively resource consuming, especially when applied to such a large corpus. More importantly, for our purposes, we need a classification which is conceptual rather than syntactic—taking into consideration not only strictly adherence to a formally defined category but also close resemblance. For example, a method which invokes another, can be rewritten without such invocation, by simple inlining and then applying local polishing. It requires a human to reveal the fact that this inlined call is in fact a case of (say) default arguments.

(On a side note, recall that dynamic analysis is not easier than static analysis. It is a great achievement to assemble a software corpus from so many components. But, it is a much higher mountain to set out a running environment for all of these components, each with its own bugs, idiosyncratic reliance on external libraries of very specific versions, and specific weird constraints on the execution environment. And, as if this is not sufficiently difficult, the question of finding "typical" inputs or "runs" has to be addressed.)

The alternative direction taken here is of a controlled human evaluation. We somewhat compromise the preciseness of the definitions, and employ humans to classify and understand the studied body of software.

Of course, it is unrealistic to apply such human analysis to large data corpus such as Qualitas. But, it is possible, as we did here, to subject a random sample drawn from the

[5] http://www.cs.auckland.ac.nz/∼ewan/corpus/

```
class Top{}
class Middle extends Top{}
class Bottom extends Middle{}

class Up{
  void f(Top t){/*...*/}
  void g(Bottom b){/*...*/}
}

class Down extends Up{
  void f(Middle m){/*...*/}
  void g(Middle m){/*...*/}
}
```

Fig. 1. Different behavior in different languages

corpus to human analysis and then use statistical methods to reason about the reliability of this analysis, and to deduce conclusions on the entire corpus, and through this, on the illusive global programming practice.

Outline. *The remainder of this paper is organized as follows. In Section 2 we describe the setting of our empirical evaluation, the results of the automatic analysis, the sampling and the employment of human evaluators. Section 3 presents the taxonomy of the kinds of overloading that may be found in actual code, as developed with the aid of the human experimenters. The reliability of the classification is studied in Section 5, which also lays the foundation for deduction of conclusions regarding the entire corpus from the sample. The results of the classification according to this taxonomy are presented in Section 6. Section 7 concludes.*

2 Research Method

This section describes the method of experimentation, both automatic and manual, and the JAVA corpus in which it was carried out.

2.1 Definitions

The Java Language Specification [12] defines method overloading as follows:

> " If two methods of a class (whether both declared in the same class, or both inherited by a class, or one declared and one inherited) have the same name but signatures that are not override-equivalent, then the method name is said to be *overloaded*."

Thus, overloading can occur between **`public`** and **`private`** methods, **`static`** and not-**`static`** methods, **`abstract`** and **`final`** methods, etc.

1. We restrict our attention to method (and constructor) overloading, even though one may argue that there are other kinds of overloading, e.g., when a class features a data member and a function member of the same name. Similarly, we exclude overloading of the '`+`' operator, the **`final`** keyword, etc.
2. Even though the JAVA semantics precludes a definition of two methods which are different only in their return type, cases of this sort can be (and indeed are) found in `.class` files, e.g., as a means for implementing co-variance by certain JAVA compilers. This *synthetic* overloading is ignored in our study.

A *constructor cohort* is the set of constructors of a class. Methods are grouped in *method cohorts*, each being the maximal set of methods sharing the same name, and available in the same user-defined type, that is a **`class`**, an **`interface`**, an **`enum`** or an annotation. In this paper we restrict attention to *non-degenerate* cohorts, i.e., cohorts with two or more *peers*.

The *primary* methods in a method cohort, are those which are first *introduced* or *reimplemented* in the type. The remaining methods, i.e., those which are *inherited* from a parent, are called *secondary*.

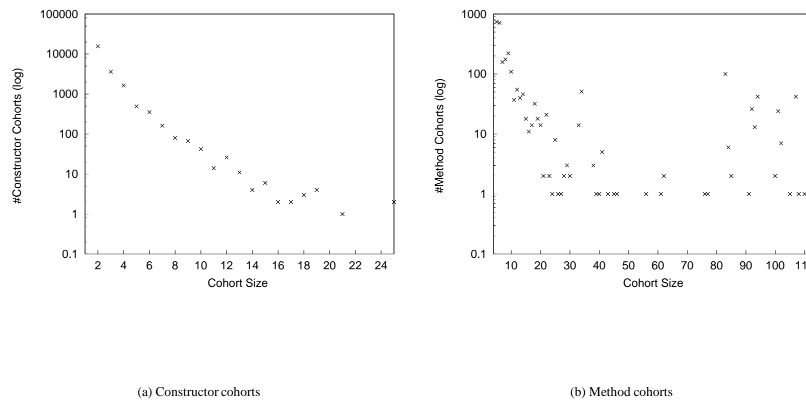
2.2 Data and Automatic Analysis

Our study started with the Qualitas corpus, consisting of 99 open-source JAVA applications, which was used extensively in the literature (e.g., in [2, 3, 17, 21, 23]) and considered to be well representing the standard programming practice in JAVA. The corpus was pruned to the most recent version of each application in it. Overall, the remaining data set consisted of 6,538 packages, with 128,482 classes, 14,214 interfaces, 48 enumerated types, and 106 annotations.

Our evaluation began with an automated analysis, which scanned the entire corpus for occurrences of overloaded methods and constructors. This analysis was carried out on the bytecode representation with a precise implementation of the above definition of overloading in the Java Tools Language (JTL) [8].

Constructor Cohorts

There were 162,495 constructors in total in the corpus. This number includes also synthetic constructors, which are generated automatically by the compiler when the programmer does not define any constructor for a given class, except when the class is *anonymous*, which do not have any constructors.



(a) Constructor cohorts (b) Method cohorts

Fig. 2. Distribution of cohorts' size (semi-logarithmic scale).

Figure 2(a), depicting the number of classes defining each number of constructors, shows the typical Zipf law distribution [24], that is $f(k) \propto k^{-\alpha}$ where $\alpha > 1$, as found in many software metrics [7]. It can be seen that the majority of all classes have only one constructor. (the exact number is 83%). We also found that 35% of constructors take part in non-degenerate cohorts. (Note that synthetic constructors never participate in non-degenerate cohorts.)

It is also evident there are a number of classes with a large number of constructors, and even two classes with as many as 25 constructors. Yet, on average the number of constructors per class is small: 1.264.

In the corpus, we found 1,030,623 method definitions (a definition of a method as **`abstract`** or in an **`interface`** being counted); this number includes redefinitions of methods. Of these, 148,192 methods are peers in 45,352 non-degenerate cohorts, i.e., slightly over 14% of the methods are overloaded. It follows that, as might be expected, overloading is much more prevalent among constructors than with methods.

Method Cohorts

Figure 2(b) is the equivalent of Figure 2(a), but focusing on method cohorts. It can be inferred from the graph that method cohorts tend to be larger than constructor cohorts. In fact, the average non-degenerate method cohort size is 3.27.

The linear decrease, typical of Zipf distribution, is not as evident here. With some imaginative effort, we can discern here a Zipf like distribution describing most cohorts' sizes (note that the slope of the Zipf decrease in constructors is much shallower than in methods), combined with a cluster of giant cohorts with over 80 methods.

Clearly, the size of this cluster exceeds what might be predicted by the Zipf distribution. A closer look at the 304 cohorts with 80 methods or more, shows that almost all of these are part of an implementation of the VISITOR design pattern [10]. In fact, the name of 268 giant cohorts is simply `visit`, while 31 cohorts are named `endVisit`. The remaining 5 giants can probably be explained by the tail of the Zipf distribution.

2.3 Sampling and Human Classification

Pre-Test

The pre-test phase was designed to produce a taxonomy of overloading, consisting of clear and unequivocal definitions, which are not merely that, but also effective for classifying concrete use of overloading in JAVA.

The development of taxonomy commenced in a brain storming session between the authors, based on our own JAVA programming experience and on sporadic inspections of cohorts found in the corpus.

This draft was then perfected using the following process: A random sample of 100 method cohorts was selected from the ensemble of such cohorts found by the automatic analysis of the corpus. The sample was restricted to cohorts satisfying the following conditions: (i) The cohort is associated with a **`class`**. That is, we excluded cohorts of **`interface`**s (no cohorts were found in **`enum`**s nor in annotations). (ii) At least one method in the cohort was non **`abstract`**. Cohorts were then further trimmed down to include only methods defined in the same class, i.e., primary overloading.

The sample was then subjected to human classification as follows. First, *cohorts* were classified by the second author, using the taxonomy draft. In the course of doing so, the taxonomy was refined, definitions were clarified, categories reorganized, etc. The refined taxonomy was then explained to three volunteer computer science graduate students, with a solid background in object oriented languages. (This explanation involved examples taken from the corpus, but not from the sample.) The raters were then asked to classify 50 specimens of the sample, 25 of which were common to all

raters, while the remaining 25 specimens were specific to the rater. (Other than these conditions, the distribution of specimens among the raters was random.)

The results were then manually inspected by the authors, using discrepancies for further refinement of the definitions and the taxonomy.

Method Cohorts Categorization

Having gained our initial confidence in our taxonomy we proceeded to experimentation concerning reliability—that is the extent at which human rating according to it is reproducible. To do so, we repeated the rating of method cohorts. This time, with 10 recruits, undergraduate- junior and senior students. All students have successfully completed the Technion's *Object Oriented Programming* course. They were each offered a monetary reward for their efforts (200 NIS, roughly equivalent to 50USD). The taxonomy and the categories in it were then explained to the raters in a two-hour frontal presentation (the presented slides are available online[6].)

A newly selected sample of 100 method cohorts was then distributed among the participants, where the random distribution satisfied the conditions that each rater was assigned 40 cohorts and that each cohort was rated by four independent raters. To encourage seriousness, raters were promised (and paid) 2.5NIS (about $0.62) for each correct categorization. The rating process lasted about 2.5 hours. It was carried out in a supervised setting, in which the raters could not communicate with each other. In addition, and independent of the student raters, all cohorts were rated by the second author.

A battery of statistical tests was then applied to the raw results of method cohorts classification. As reported below, these tests indicated that the rating of cohorts by the second author is reliable with high confidence margins.

Constructor Cohort Classification

Relying on the reliability of the classifications of the second author, we did not repeat the same process for constructor cohorts classification, instead, this classification was done solely by the second author. The sample consisted again of 100 cohorts selected at random from the constructor cohort base.

3 Taxonomy of Overloading

We now present the fruit of the experiments and process of perfecting a taxonomy of the use of overloading in JAVA. This section gives a high level survey of the main categories. The next section elaborates, describing the specific patterns in greater detail.

The primary question that our classification asks in considering an onverloading incidence is how coincidental it is. An extreme case is, for example, a class representing a cartoon cowboy, featuring an overload of method `draw`. At the other end, will find e.g., the overloaded method `setLocation` of class `awt.Point`, whose partial view is presented in Figure 3.

[6] http://www.cs.technion.ac.il/∼ssdl/pub/JavaMethodClassification/JavaOverloadingClassification.pdf

```
class Point {
  public int x;
  public int y;

  public void setLocation(Point p) {
   setLocation(p.x, p.y);
  }

   public void setLocation(int x, int y) {
    move(x, y);
   }

   public void move(int x, int y) {
    this.x = x;
    this.y = y;
   }
}
```

Fig. 3. An example of INTRINSIC overloading

A classification according to this criterion is important since overloading is criticized precisely because there is no enforcement of any semantical relationship between methods in the same cohort. Yet, in practice we find that such methods are often related, and that overloading is often used to capture a situation in which the input to a certain operation can be presented in different ways. Figure 4 draws a spectrum of the relationship between the semantics of overloaded methods.

The boxes in the figure represent *overloading categories* (which we will interchangeably also call *patterns*).

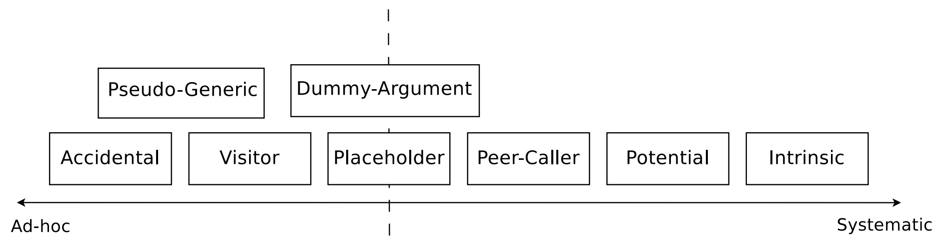


Fig. 4. The overloading spectrum, from systematic to ad hoc.

As we move to the right from the central dividing line we meet patterns which are progressively more systematic, that is, patterns in which the semantics of peers is progressively more related. Conversely, a move to the left reveals patterns in which

overloading is of a more ad hoc nature. Boxes on the central line represent "neutral" patterns, i.e., patterns in which overloading can be *either* ad hoc or systematic.

Systematic overloading occurs e.g., when the body of one overloaded method is in essence a transformation of its arguments, followed by a call to one of its cohort peers. Cases of this sort fall into the INTRINSIC category (this category includes also other kinds of overloading, as explained later). The POTENTIAL category is similar in that our human reviewers concluded that it can be brought into the INTRINSIC category with minimal effort. PEER-CALLERS are overloading instances in which a method calls its peer, but it is not clear whether it can be rewritten in the INTRINSIC form.

At the other end of the spectrum, the ACCIDENTAL category, refers to cases in which no peer calls occurred, and no other relationship between peers could be identified.

On the dividing line, we find, PLACEHOLDERS in which all methods in the cohort have no body. The overloading kind can fall into any other category, depending on the implementation in the inheriting class or classes. Cohorts of this patterns were exculded from the sample since their classification is trivial. On this line, we also find the rather rare DUMMY ARGUMENT in which an extra, otherwise unused, argument distinguishes between peers (particularly constructors) which need the same arguments' type sequence. (Think for example on distinguishing between a polar- and cartesian- based constructors to a class `Point`).

Notice that the above categories apply to a pair of peers. Different pairs selected from the same cohort, do not necessarily fall into the same category. An exception is the VISITORS category, which represents the use of overloading for realizing the VISITOR design pattern. Usually, in a cohort which is classified into the VISITORS category, most, if not all, peers fall into this category.

Finally, the PSEUDO-GENERIC category pertains to cases of use of overloading in JAVA which were candidates to generic based implementation, had JAVA generics been applicable to primitive types, e.g., as in the different implementations of `Math.round` for types **`double`** and **`float`**. Here again, we may expect several peers to fall into this category.

4 Overloading Patterns Catalog

In this section we discuss the overloading patterns in greater detail and exemplify their use. Our presentation starts from the systematic end of the overloading spectrum and progresses towards the ad-hoc patterns.

4.1 INTRINSIC Overloading

The INTRINSIC category refers to methods whose relationship with its name-peer is semantical. Further breakdown of this category is offered by Figure 5.

In the figure we see that there are two main subcategories here: RESENDING in which defines an asymmetric relation between two methods in a cohort and INDUCED applies equally to all of the methods in a cohort

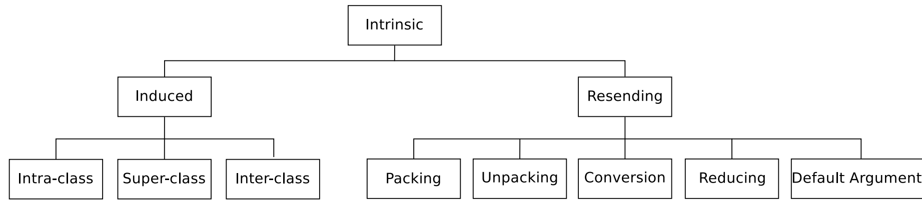


Fig. 5. Classification of intrinsic overloading patterns.

Induced Overloading

In the INDUCED category, overloading in one cohort *induces* an overloading in another cohort. Suppose that the designer of a certain class sees it fit to equip this class with two constructors. Now, if this class is extended by way of inheritance, then it is only natural that the subclass will offer two constructors, each delegating to a distinct constructor in the base class. This situation occurs in many other situations, e.g., in design patterns [10] such as COMPOSITE and DECORATOR, and in general, in all cases in which a class delegates duties to another. In all of these, the desire to provide a rich and consistent interface brings about overloading (in the delegator) which replicates overloading in another cohort (the delegate).

The requirement in our natural language description of this category was double folded: *(i)* the delegator and its delegate have identical argument list; *(ii)* the delegator invokes the delegate precisely once in any of its execution paths.

The left hand side of Figure 5, shows a breakdown of INDUCED based on the relationship between the delegator and the delegate:

1. INTRA-CLASS DELEGATION, in which each method invokes a method with a different name but same arguments of the same class, and in the case the callee is not **static**, the call is to **this**. One example of this pattern is the cohort named `removeBundle` in Eclipse's class `RequireBundleHeader` which is located in package `org.eclipse.pde.internal.core.text.bundle`:
```
void removeBundle(String id) {
  removeManifestElement(id);
}

void removeBundle(RequireBundleObject bundle) {
  removeManifestElement(bundle);
}
```
2. SUPER-CLASS DELEGATION type in which each method invokes a method with the same signature on the **super** class. This pattern is common in constructors, as can be found in class `RuntimeException` of the JAVA standard library:
```
public class RuntimeException extends Exception {
  public RuntimeException() {
     super();
  }
```

```
  public RuntimeException(String message) {
    super(message);
  }

  public RuntimeException(String message, Throwable cause) {
    super(message, cause);
  }

  public RuntimeException(Throwable cause) {
    super(cause);
  }
}
```

3. INTER-CLASS DELEGATION, in which each method invokes a method with the same signature on a member object. The cohort named `updateString` in class `AS400JDBCRowSet` (found in package `com.ibm.as400.access`) drawn from the open source version of the IBM toolbox for Java (JTOpen) demonstrates this pattern:

```
public class AS400JDBCRowSet implements RowSet, Serializable {
  /* . . . */
  private AS400JDBCResultSet resultSet_;
  /* . . . */
  void updateString (int columnIndex, String columnValue) {
    validateResultSet();
    resultSet_.updateString(columnIndex, columnValue);
    eventSupport_.fireRowChanged(new RowSetEvent(this));
  }
  void updateString (String columnName, String columnValue) {
    validateResultSet();
    resultSet_.updateString(columnName, columnValue);
    eventSupport_.fireRowChanged(new RowSetEvent(this));
  }
}
```

Resending

In the RESENDING category, one overloaded method carries out its mission by resending its arguments to its peer after some preprocessing phase. We say that a designated caller method is RESENDING to a designated callee method when all four of the following conditions hold: *(i)* the caller invokes the callee precisely once in any of its execution paths, *or* there is a single call site, which is executed iteratively; *(ii)* the caller does not call any other peer; *(iii)* the returned type of the caller and the callee is the same; and *(iv)* if the caller returns a value, it is the value returned by callee, unaltered.

Figure 5 distinguishes between five patterns of RESENDING, based on the processing work carried out by the caller on the arguments it passes on to the callee:

1. PACKING, in which the caller packs some of its arguments into a collection or an array and then sends it to the callee. Method `setValue` of class `PreferenceConverter`

of found in package `org.eclipse.jface.preference` of the Eclipse development environment, illustrates this pattern:

```
void setValue(IPreferenceStore store, String name, FontData value) {
  setValue(store, name, new FontData[] { value });
}
```

2. UNPACKING, in which the caller accepts a collection or an array, and invokes the callee on each element of the collection (array). One example of this pattern is method `convertToVector` of class `DefaultTableModel` which is located in package `javax.swing.table`:

```
Vector convertToVector(Object[][] anArray) {
  if (anArray == null)
   return null;
  Vector v = new Vector(anArray.length);
  for (int i=0; i < anArray.length; i++)
    v.addElement(convertToVector(anArray[i]));
  return v;
}
```

3. CONVERSION, in which the caller converts one or more of its arguments to another type, to make it suitable for the callee to digest. Method `setLocation` of class `Point` depicted in Figure 3 is a case of this pattern.
4. REDUCING, in which the caller processes some of its arguments and sends a subset of the arguments to the callee, as is demonstrated by the `create` method of class `BidiOrder` which resides in package `com.ibm.as400.access` of the Azureus application:

```
ResourceDownloader create(URL url, boolean force_no_proxy) {
 ResourceDownloader rd = create(url);
 if (force_no_proxy && rd instanceof ResourceDownloaderURLImpl)
   ((ResourceDownloaderURLImpl)rd).setForceNoProxy(force_no_proxy);
 return rd;
}
```

5. DEFAULT ARGUMENTS in which overloading is used as a substitute to default arguments mechanism, and the caller does nothing but resend all of its arguments, as well as some other default value or values, to the callee. The following constructors, which belong to class `Point`, presented in Figure 3 fall into this category:

```
public Point() {
 this(0, 0);
}

public Point(int x, int y) {
  this.x = x;
  this.y = y;
}
```

4.2 POTENTIAL Overloading

The refinement of the INTRINSIC category as presented in Figure 5 is appicable in principal also to the POTENTIAL category. However, the manual task of identifying

"Potential Induce" overloading, that is, checking whether a certain method pair *could be rewritten* by delegation to another such pair, which could be anywhere in the system, is formidable. We did not ask our raters to do that, and instructed them to concentrate in finding cases in which one method could be rewritten in terms of a name-peer, whereby restricting the breakdown of the POTENTIAL category into the various sorts of RESENDING.

For example, the first version of `setLocation` method in Figure 3, rewritten as

```
public double setLocation(Point p){
  move(p.x, p.y);
}
```

belongs to this category.

4.3 PEER-CALLER Overloading

A method is classified as PEER-CALLER when it invokes one of its peers, but it is not classified as RESENDING.

Method `findResources`, drawn from class `StandardPluginClassLoader`, which resides in package `org.java.plugin.standard` illustrates such a case:

```
public Enumeration findResources(final String name) {
   List result = new LinkedList();
   findResources(result, name, this, null);
   return Collections.enumeration(result);
}
```

Since the return type of this method differs from that of the invoked peer it cannot be rewritten as an instance of the INTRINSIC category.

4.4 VISITOR Overloading

The VISITOR design pattern is a way of separating operations from the data structure upon which they operate. This pattern is often realized in JAVA by an interface which has a `visit()` method for each class whose objects may reside in the data structure. Thus, this design pattern usually implies using overloading. Moreover, often the data structure may contain objects of many different type, and as a result the number of overloaded `visit()` methods becomes very high.

One example of this category is the cohort named `visit` of class `GenericVisitor`, found in package `org.eclipse.jdt.internal.core.dom.rewrite` of the Eclipse framework. There are 83 methods in this cohort, each accept a single parameter, which represents a type of a node in the Abstract Syntax Tree representation of a JAVA program.

5 Statistical Analysis of the Experiment

The definitions of the various categories in the previous section may seem more intuitive than precise, and difficult to formalize. Take for example the definition of the DEFAULT ARGUMENTS category, which required that the "*caller does nothing but resend all of*

its arguments". Then, it is not difficult to construct spiteful cases, which challenge the accuracy and even decidability of, say, the phrase, "does nothing"—by adding involved computation, which would take tremendous resources for a statistical analyzer to prove vacuous, or by throwing in a reduction of the halting problem.

Instead of doing so, we restrict our classification to the intuitive meaning, as interpreted by humans, and devote this section to the reliability of this classification. We start with some general figures. Recall that raters where rewarded based on their hit rate, which was computed against the independent rating carried out by the second author. Scores ranged between 73% and 85%, and averaged at 79%.

As expected, there were many disagreements regarding classifications in the somewhat loosely defined POTENTIAL category. It turned out that *almost all* disagreements were with regard to this category. In contrast, the DEFAULT ARGUMENTS category raised the fewest disagreements, reaching a fully unanimous vote casted in 85% of the cases.

The more important question which we explore next is the systematic *statistical reliability* of this human classification. The analysis here shall demonstrate that the results of the manual classification are indeed reproducible, and that the numerical value that will be presented in the *following* section are therefore significant. At the end of the current section we remind the reader the notion of *confidence interval*, which should help in deducing conclusions regarding the entire corpus from what was observed for the sample.

5.1 Reliability of Human Classification of Overloading

Cronbach's α-coefficient [9], or for short α, is a statistic which is used in social sciences to estimate the internal consistency of multiple items within a scale. It is employed in cases, such as ours, in which the scale is nominal rather than ordinal or rational. The value of this estimator ranges between 0 and 1, where a value of 0 corresponds to the case that items are uncorrelated, i.e., all variation is due to random fluctuations. A value of 1 corresponds to the case that the items are in complete correspondence. For research purposes it is customary to require $\alpha \geq 0.8$ [19].

Cohen's κ-coefficient [6], or for short κ, is a leading measure of agreement which assesses the extent to which *two* raters give the same ratings to the same objects, while factoring out the probability of agreement between the raters that would be expected due to chance. The κ values range from -1, which means perfect disagreement to 1, meaning perfect agreement, where 0 is interpreted as agreement achieved by chance. A value of 0.6 or higher is considered as a strong agreement.

Table 1 presents the values of these two statistics in relation to the classifications carried out by the human raters in the experiment. The first row corresponds to evaluations of all cohorts which fell in the sample, while the second is restricted to cohort samples of size 2.

Examining the table, we see that the high hit rate the evaluators achieved is far from being accidental, and it cannot be attributed to chance. Further, we see high correspondence not only in classification according to top level categories, but also to sub categories.

Tab. 1. Statistical estimators of manual classifications of overloading

Experiment	Categories	Cronbach α	Cohen's κ (averaged)
All cohorts	all categories	0.877	0.731
	top-level categories only	0.886	0.743
Size-2 cohorts	all categories	0.890	0.765
	top-level categories only	0.892	0.768

Few words are in place in order to explain the method of computation, which involved data aggregation. Recall that ten raters participated in the experiment, each classifying a subset of 40 methods, while there was no single pair of raters who classified the same subset. However, calculating the α measure requires that all raters refer to the same items. Thus, instead of considering a model in which 10 raters evaluated 40 items each, we switched to a model in which we consider only the ratings of each item, without taking into consideration who rated it. Our transformed model therefore included four sets of ratings, each referring to 100 cohorts.

Although the α measure is not originally designed to check inter-rater reliability, there is evidence showing that its use for such purposes is adequate, and even desirable when multiple raters are involved [16]. Following transposing of the data set, this statistic was used to estimate the correspondence between all ratings, rather than raters.

We used the same aggregated model for κ calculations as well. The κ measure was used in this study to estimate inter-rater reliability between the second author and each of the aggregated rating sets. The table displays the average value of the four κ values that were obtained.

Finally, we should say that the excellent values reported in Table 1 are relevant only to the categories which were actually presented in the sample. As we shall see below, some of the categories in the taxonomy, although theoretically interesting, did not manifest in the sample.

5.2 Binomial proportion confidence interval

Now that we have established the statistical significance of the manual classification, it remains to determine what can be inferred about the entire corpus from the classification of the specimens in the sample. Suppose that a fraction of size p of the elements in a sample fell into a certain category, then, we would like to find a value Δp such that there is a vanishing probability that the true fraction of cases in the corpus is not within between $p - \Delta p$ and $p + \Delta p$.

The *binomial proportion confidence interval* provides this information precisely. It uses the proportion estimated in a statistical sample and allows for sampling error. There are several ways to compute a confidence interval for a binomial proportion. We chose the Wilson score interval [22] due to the good properties of this test for even a small number of trials or an extreme probability. To estimate the sampling error we calculated 95% confidence intervals using Wilson score method for a binomial proportion.

Tab. 2. The 95% confidence intervals for a sample of size 77 (confidence interval is symmetric for p and $1 - p$).

Proportion in Sample	0%	1%	5%	10%	20%	35%	50%
Confidence Interval $n = 77$	0%–6%	0%–8%	1%–13%	5%–19%	12%–30%	25%–46%	39%–61%
Confidence Interval $n = 65$	0%–8%	0%–9%	1%–15%	4%–20%	12%–32%	25%–47%	38%–62%

We calculated the confidence intervals based on a sample of size 77 (method cohorts of size two) and 65 (constructor cohorts of size two), for various proportions in the sample. The results are presented in Table 2. As can be seen in the table, the values of Δp are quite large, but still, if a certain pattern is infrequent in the sample, it is with very high probability infrequent in the corpus. Conversely, patterns which are common in the sample, are very likely to be common in the corpus.

6 Results

Now that we have established the reliability of our classification system and understood what can be inferred from its values to the full corpus, it is time to present the actual results of this classification.

6.1 Method Cohorts

Table 3 shows the distribution of sizes of cohorts that fell in the sample of 100 cohorts. As expected, a number of large cohorts were sampled. Even though the small cohorts, with only two methods, were 77% of the samples, the methods in these were 64% of the sample.

Tab. 3. Distribution of cohorts' sizes in the 100 method cohorts sample

Size	2	3	4	5	6	7	**Total**
# Cohorts	77	13	6	1	2	1	*100*
# Methods	154	39	24	5	12	7	*241*
Fraction	64%	16%	10%	2%	5%	3%	*100%*

Table 4 provides the results of manual classification of these cohorts. The numbers in the table represent the results of the classification of the second author. Note that a cohort with more than two peers could fall into several categories. In the table, a cohort was counted in a certain category if the pattern occurred in it at least once. Categories whose number of occurrences is zero are omitted.

In addition to the counts depicted in the table, 2 cohorts were such that none of the methods were implemented, i.e., PLACEHOLDERs, 13 were classified as ACCIDENTAL

Tab. 4. Manual classification of the 100 method cohorts in the sample (zero values are omitted).

Category	Sub-category	Sub-sub-category	# Cohorts
Intrinsic (68)	Resending (54)	Default arguments	28
		Conversion	17
		Reduction	5
		Packing	3
		Unpacking	1
	Induced (14)	Inter-class delegation	10
		Intra-class delegation	3
		Super-class delegation	1
Potential (18)	Resending (18)	Default arguments	9
		Conversion	8
		Unpacking	1

since none of their methods invoked any of their cohort peers (or could be implemented as such), and 6 cohorts contained methods which invoke each other, but did not match any of the patterns and were therefore classified as PEER-CALLERS. The sample did not include any instances of DUMMY ARGUMENT, VISITOR and PSEUDO GENERIC.

The table reveals a strong tendency towards systematic rather than ad hoc use of overloading: More than half of the cohorts involve a resending pattern. The most frequent pattern of overloading that was observed is that of default parameters, which was observed in 28% of the cohorts and has the potential of being implemented in additional 9%.

6.2 Method Pairs

The quadratic increase in the number of pairs of peers makes it difficult to analyze the patterns of use of overloading in cohorts with more than 2 methods. Worse, inspecting in isolation all possible pairs in a cohort is likely to produce confusing information which may need a bit of pondering before the underlying structure of the cohort can be revealed.

Consider for example the `fill` cohort in class `Arrays`, which has 18 methods (2 for each of JAVA's primitive type and 2 for `Object`) and 153 different pairs. These pairs can be broken down as follows. The 9 pairs of the sort of

$$\left\langle \texttt{fill(}\textbf{float}\texttt{[],}\textbf{int}\texttt{,}\textbf{int}\texttt{,}\textbf{float}\texttt{)}, \texttt{fill(}\textbf{float}\texttt{[],}\textbf{float}\texttt{)} \right\rangle$$

are DEFAULT ARGUMENTS; (replacing **`float`** in any other primitive type or in `Object`). The 36 pairs of the sort of

$$\left\langle \texttt{fill(}\textbf{float}\texttt{[],}\textbf{int}\texttt{,}\textbf{int}\texttt{,}\textbf{float}\texttt{)}, \texttt{fill(}\textbf{char}\texttt{[],}\textbf{int}\texttt{,}\textbf{int}\texttt{,}\textbf{char}\texttt{)} \right\rangle$$

(where **`float`** and **`char`** can be replaced likewise) are PSEUDO GENERIC. And, the 36 pairs of the sort of

$$\left\langle \texttt{fill(}\textbf{byte}\texttt{[],}\textbf{byte}\texttt{)}, \texttt{fill(Object[],Object)} \right\rangle$$

Tab. 5. Results of manual classification of 77 method cohorts of size two (zero values are omitted).

Kind	Sub-kind	Pattern	# Cohorts
Intrinsic 57% (44)	Resending 44% (34)	Default arguments	19
		Conversion	9
		Reduction	3
		Packing	2
		Unpacking	1
	Induced 13% (10)	Inter-class delegation	10
		Intra-class delegation	0
		Super-class delegation	0
Potential 19% (15)	Resending 19% (15)	Default arguments	7
		Conversion	7
		Unpacking	1
Accidental 13% (10)			10
Placeholders 3% (2)			2
Peer-Callers 8% (6)			6

are also are PSEUDO GENERIC. The remaining 72 pairs, which constitute 47% of the lot, do not fit into any of the categories.

Assigning appropriate weights to the different categories for cohorts with more than 2 peers can be complicated, But, even if this hurdle is overcome, requiring the human raters to reveal the underlying structure would not only have complicated the experiments, but also introduced unnecessary noise. Therefore, our more in depth analysis was restricted to the 77 cohorts in the sample which were of size 2. Table 5 provides a breakdown of the classification of these cohorts. Unlike Table 4, each cohort occurs precisely once in this table

As in Table 4, the most common pattern is that of DEFAULT ARGUMENTS, being used by 24.6% of the cohorts, with additional 9% which have the potential of using it. Again, we see strong tendency towards systematic use of overloading. This tendency is further depicted visually in Figure 6, which portrays a histogram of the breakdown into top level categories.

We see that 84% of the total weight falls in the right hand side of the figure. Six out of seven of the pairs in our sample exhibit systematic overloading, and three out of five fall in the INTRINSIC category. Also, summing up the values in Table 5 we determine that in three out of five pairs of overloaded methods, one method calls another. Relying on the confidence intervals summarized in Table 2, we can further infer that with high probability these estimates apply to the full corpus, to within a $\pm 10\%$ margin.

Finally, we remark that the tendency towards more systematic use of overloading should increase, or at the least stay the same, as the cohort size increases. This is of course with the exception of visitors, in which even though the intended semantics is similar, it is unclear whether the actual implementation of different visitors is likely to

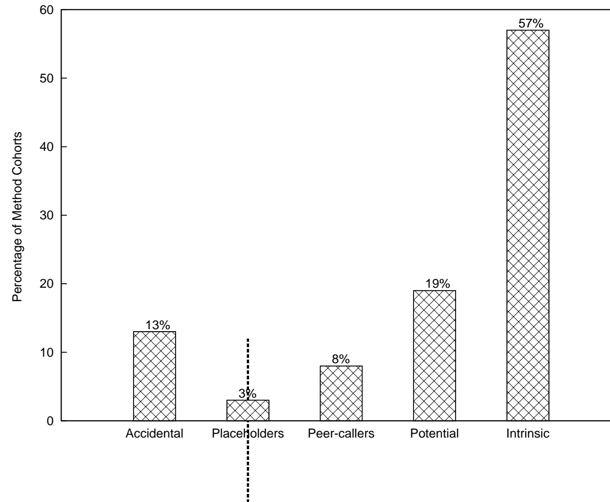


Fig. 6. Spectrum of systematic overloading in sampled method cohorts of size 2.

show systematic repetition. Luckily, visitors are very rare, and we can therefore conclude that the use of overloading in the vast majority of cases is very systematic, and that programmers are not tempted to abuse this language feature.

6.3 Constructor Cohorts

Table 6 displays the results of manual classification of the 100 cohorts in the constructors sample.

Tab. 6. Results of manual classification of 100 constructor cohorts.

Kind	Sub-kind	Pattern	# Cohorts
Intrinsic (59)	Resending (34)	Default arguments	25
		Conversion	8
		Reduction	1
		Packing	0
	Induced (25)	Inter-class delegation	0
		Intra-class delegation	3
		Super-class delegation	22
Potential (13)	Resending (13)	Default arguments	12
		Conversion	1

As in Table 4, we did not count each pair of overloaded constructors separately, instead, each cohort was counted once for each pattern that occurred at least once. In 32 constructor cohorts no particular pattern was identified between any of the pairs, and hence, the entire cohort is classified as ACCIDENTAL.

No constructor cohort was classified as vanilla PEER-CALLER. In other words, there was no constructor cohort in which a constructor invokes other constructor and does not match a more specific pattern. This is not very surprising, since the language syntax mandates that the inter-constructor invocation must be the first statement. The only allowed computation, computing the actual arguments prior to the invocation, does not admit much programming freedom or creativity.

Examining the table further, we see a clear tendency towards more systematic use of overloading. But, in comparison with Table 4, it is evident that this tendency is not as forceful as it is with methods.

6.4 Constructor Pairs

In order to appreciate more accurately the tendency towards systematic overloading in constructors, we now concentrate, as we did with methods, in cohorts of size 2. Table 7 shows the distribution of sizes in the sample of constructor cohorts. We see that still, a substantial portion of the constructors fell in the first column of the table.

Tab. 7. Distribution of cohorts' sizes in the 100 constructor cohorts sample (zero values are omitted)

Size	2	3	4	5	6	7	8	**Total**
# Cohorts	65	21	9	1	1	2	1	*100*
# Methods	130	63	36	5	6	14	8	*262*
Fraction	50%	24%	14%	2%	2%	5%	3%	*100%*

Table 8 presents a view of the results which contains classifications of constructor cohorts of size two.

Comparing the results of constructors classifications to those of methods classifications it can be observed that INDUCED overloading is more frequent in constructors than in methods. and that the one of the most common pattern is SUPER CLASS DELEGATION. Again, this is what may be predicted by the language syntax: a constructor with a non-empty parameters list in a base class often induces a constructor with the same parameters list in its derived classes.

Finally, Figure 7 summarizes the spectrum of use of overloading in restricted sample. The tendency towards systematic use of overloading is evident, but it is also clearly weaker than in methods (Figure 6).

We see that about three out of five cohorts with two constructors exhibit systematic overloading at some level, and about one in two is a resend to peer. Again, we have the same reasons to believe that this tendency is not spoiled as we move to larger cohorts.

Tab. 8. Results of manual classification of 65 constructor cohorts of size two (zero values are omitted).

Kind	Sub-kind	Pattern	# Cohorts
Intrinsic 46% (30)	Resending 26% (17)	Default arguments	13
		Conversion	4
		Reduction	0
		Packing	0
	Induced 20% (13)	Inter-class delegation	0
		Intra-class delegation	0
		Super-class delegation	13
Potential 17% (11)	Resending 17% (11)	Default arguments	10
		Conversion	1
Accidental 37% (24)			24

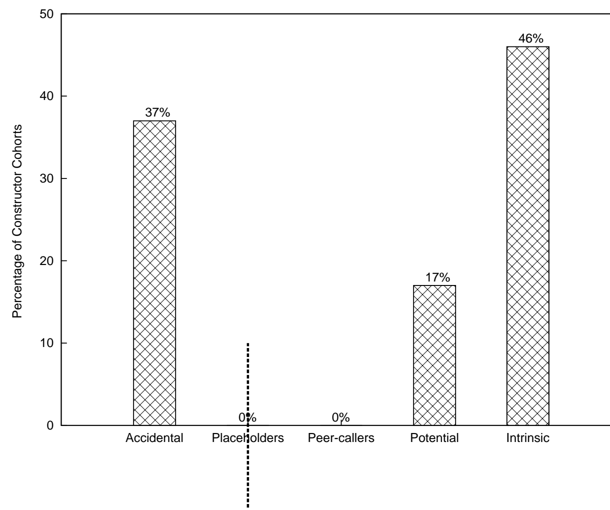


Fig. 7. Spectrum of systematic overloading in sampled constructor cohorts of size 2.

7 Conclusions and Further Research

7.1 Summary of Results

We found that overloading is used extensively in the corpus: 35% of all constructors and 14% of all methods. Cohorts tend to be larger in methods, with an average cohort size slightly greater than 3, while the average number of constructors per class is about 1.3. The distribution of cohort sizes is Zipf-like, except that method cohorts feature cluster of large cohorts attributed to the VISITOR design pattern.

We developed a taxonomy for the classification of the use of overloading in actual JAVA programs. The taxonomy was refined and exacted in a process involving two stages of subjecting sized samples to human rating in a controlled environment. The reliability of the classification was validated, at least for the major categories, by engaging statistical tests traditionally used in social sciences.

Statistical analysis also showed that at least six out of seven cases of use of method overloading are more systematic than ad hoc. The fact that overloading is mandatory in the definition of multiple constructors probably explains our finding that the systematic overloading is somewhat less frequent in constructors, occurring in about three out of five cases. These results (whose error margin is about 10%) may answer the allegation that overloading is likely to be abused.

It was determined that the most frequent use of overloading is for simulating defaults arguments. This use pattern occurring in about a quarter of overloaded methods, while additional 10% of these can probably be rewritten as such. Similarly, about a third of the cases in which overloading is used with constructors, are, or can be, expressed as a form of overloading.

7.2 Further Research

It is interesting to study the smaller categories which were not captured by our sample. This can be done e.g., by employing refined sampling techniques.

We are intrigued by the INDUCED category, which suggests that overloading is viral—the use of overloading in one class leading to overloading in another. There may be room for checking whether generics could address this duplication.

Acknowledgments. We thank Itay Maman for his thoughtful comments, and pay great tribute to Irit Hershkowitz for her advices on the statistical analysis.

References

1. K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, Reading, Massachusetts, 1996.
2. R. Barker and E. D. Tempero. A large-scale empirical comparison of object-oriented cohesion metrics. In *APSEC*.
3. G. Baxter, M. Frean, J. Noble, M. Rickerby, H. Smith, M. Visser, H. Melton, and E. Tempero. Understanding the shape of java software. *SIGPLAN Not.*, 41(10):397–412, Oct. 2006.
4. A. Beugnard and S. Sadou. Method overloading and overriding cause distribution transparency and encapsulation flaws. *Journal of Object Technology*, 6(2):31–46, 2007.
5. L. Cardelli and P. Wegner. On understanding types, data abstractions, and polymorphism. *ACM Comput. Surv.*, 17(4):471–522, Dec. 1985.
6. J. Cohen. A coefficient of agreement for nominal scales. *Educational and Psychological Measurement*, 20, 1960.
7. T. Cohen and J. Gil. Self-calibration of metrics of Java methods. In *Proc. of the 37th Int. Conf. on Technology of OO Lang. and Sys. (TOOLS'00 Pacific)*.
8. T. Cohen, J. Y. Gil, and I. Maman. JTL—the Java tools language. In P. L. Tarr and W. R. Cook, eds., *Proc. of the 21st Ann. Conf. on OO Prog. Sys., Lang., & Appl. (OOPSLA'06)*.

9. L. Cronbach. Coefficient alpha and the internal structure of tests. *Psychometrika*, 16(3):297–334, 1951.
10. E. Gamma, R. Helm, R. E. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts, 1995.
11. J. Gil and I. Maman. Micro patterns in Java code. In R. Johnson and R. P. Gabriel, eds., *Proc. of the 20th Ann. Conf. on OO Prog. Sys., Lang., & Appl. (OOPSLA'05)*.
12. J. Gosling, B. Joy, G. L. J. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, Reading, Massachusetts, 3rd ed., June 2005.
13. A. Hejlsberg, S. Wiltamuth, and P. Golde. *The C# Programming Language*. Addison-Wesley, Reading, Massachusetts, 2nd ed., Oct. 2003.
14. ISE. *ISE EIFFEL The Language Reference*. ISE, Santa Barbara, CA, 1997.
15. B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice-Hall, 2nd ed., 1988.
16. R. N. MacLennan. Interrater reliability with spss for windows 5.0. *The American Statistician*, 47(4):292–296, Nov. 1993.
17. H. Melton and E. Tempero. Static members and cycles in java software. In *ESEM*.
18. B. Meyer. Overloading vs. object technology. *Journal of Object-Oriented Programming*, pp. 3–7, 2001.
19. J. Nunnally and I. Bernstein. Psychometric theory. *rds.epi-ucsf.org*, 1978.
20. B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, Massachusetts, 3rd ed., 1997.
21. E. Tempero, J. Noble, and H. Melton. How do java programs use inheritance? an empirical study of inheritance in java software. In J. Vitek, ed., *Proc. of the 22nd Euro. Conf. on OO Prog. (ECOOP'08)*.
22. E. B. Wilson. Probable inference, the law of succession, and statistical inference. *Journal of the American Statistical Association*, 22(158):209–212, 1927.
23. H. Y. Yang, E. Tempero, and H. Melton. An empirical study into use of dependency injection in java. In *ASWEC '08: Proceedings of the 19th Australian Conference on Software Engineering*.
24. G. K. Zipf. *The Psychobiology of Language*. Houghton-Mifflin, New York, NY, USA, 1935.