On the Use of Properties in Java Applications

Markus Lumpe, Samiran Mahmud, and Rajesh Vasa
Faculty of Information and Communication Technologies
Swinburne University of Technology
Hawthorn, VIC3122, Australia
{mlumpe,smahmud,rvasa}@swin.edu.au

***Abstract*—When building software systems, developers have to weigh the benefits of using one specific solution approach against the risks and costs of using another one. This process is not random. Certain preferences, architectural styles, and solution domain pressures create systematic biases that we can measure in order to assess their impact on the system being built and the underlying development process itself. In this paper we explore, whether the getter and setter methods in Java give rise to a bias also. Getter and setter methods, called "properties", are perceived commonplace and considered by some as a threat to data encapsulation. However, little empirical evidence exists that can reliably inform us about the real impact of the use of properties in Java. For this reason, we examined 102 open-source Java systems and discovered that properties are employed much more carefully than one might expect. Contrary to some folklore, developers use properties not just to gain access to an object's private state, but in a systematic and responsible manner and, in general, consistent with the domain requirements of the developed software system.**

***Keywords*-empirical study, open-source software, Gini coefficient, decision framing**

I. INTRODUCTION

Data encapsulation, the controlled access to the state of object instances, constitutes one of the fundamental tenants in the object-oriented programming paradigm [1]. In its purest form, data encapsulation is achieved by rendering the visibility of an object's state *private* and allowing access to and manipulation of it only through a well-defined set of *public* methods [2], [3]. But do developers actually follow this principle? Moreover, do certain architectural styles and design guidelines yield software artifacts with less pronounced object-oriented data encapsulation?

These questions are at the center of a recent study by Tempero [4], who analyzed the *use of fields* in the *Qualitas Corpus* [5], a collection of open-source Java systems. Tempero discovered that most systems in the corpus contain non-private fields, which may be taken as an indicator for a systematic breach of data encapsulation in those systems. However, the actual number of exploits is much smaller. Only 12% of the exposed classes [4] are subject to non-private field access. Tempero stipulates that non-private fields may be a result of accidents (or oversights) rather than conscious design decisions as the studied systems do not take advantage of non-private fields.

But there is another dimension of data encapsulation that we need to consider also – *getters* and *setters* – designated member functions to access and manipulate the state of objects. Getters and setters are collectively called *properties* and have been popularized in the nineties, when they were introduced as a new linguistic element to Borland's *Delphi* object model [6]. Though possible, the purpose of getter and setter methods is not to circumvent the visibility modifiers, in particular *private*, but to provide a means to effectively convey the intent of code while at the same time giving developers the tools to retain the level of protection generally associated with information hiding.

The Java language [7] lacks built-in support for the property mechanism. Instead, developers have to adhere to a coding convention or *architectural style* [8], which requires that a property is to be signified by two components [9], [10]: the field name associated with the property and the prefix "get" and "set" to denote a getter and setter, respectively. For example, the method name `getColor` stands for a getter for field `Color`. Similarly, `setColor` denotes the setter for field `Color`.

Coding conventions, like Java's *naming pattern* for getter and setter methods, give rise to a systematic bias towards the way developers express intent in software. But how strong is this bias in actual software systems? Do the Java-specific idioms change the developers' perceptions and, consequently, do developers become hence less inclined to preserve data encapsulation? Moreover, how frequent is the use of getter and setter methods in Java systems? The answers to those questions will help us in establishing an understanding of *how* developers use the property mechanism in particular and develop software using Java in general.

A well-established approach to tackle such questions is to conduct an empirical study, collect suitable software metrics, and perform an associated data analysis and interpretation [11]. There is, however, one complicating factor. Software metrics data is, in general, heavily skewed [12], [13], an aspect that hampers precise inference using standard descriptive statistical analysis. Central tendency measures like "average" or "mean" only work, if the measured population has a *Gaussian* distribution. Moreover, variations in size and amplitude of individual values render comparative analysis of two or more populations increasingly unreliable. But

without the proper means to compare metrics data, we cannot gather any meaningful information about the studied software systems.

Vasa et al. [14] have recently proposed a new analysis method that uses the *Gini coefficient* [15] as a summary measure to effectively capture the nature of an underlying, heavily skewed, software metrics distribution. The Gini coefficient is a single number between 0 and 1, in which 0 denotes *perfect equality* (*i.e.*, the allocation of features adheres to an even pattern) and 1 stands for *perfect inequality* (*i.e.*, there is but one *God*-class in the system). Software analysis based on Gini coefficients has proven well-suited for the interpretation of software metrics data [14]. In addition, the Gini coefficient-based technique offers a more reliable method to recover and better understand developer decisions than central tendency measures.

In this paper we apply this new analysis technique to study the getter and setter methods usage profiles in Java software. We use as our reference data set the *Qualitas Corpus* [5]. However, some of the systems in the corpus are rather small. In order to improve the statistical significance, we decided to consider only systems with more than 150 classes. As a consequence, at least 16 systems from the *Qualitas Corpus* do not meet our selection criterion. To compensate for this loss, we searched the Internet and found 18 suitable replacement systems with tiles-2.1.2 being the smallest and cayenne-2.0.4 being the largest system. As a result, our experimental data set consists now of 102 systems totaling $153,553$ classes, $1,037,326$ methods, $335,729$ fields, $231,841$ getter, and $101,358$ setter methods.

Using this data set, we run the corresponding metrics data extraction and observed the following:

- Getter and setter methods, both, are very unevenly distributed in software systems, but to varying degrees. However, developers prefer to define these methods in a relatively small number of classes.
- There is no empirical evidence that the use of getter methods is a function of the solution design or domain. But, the same does not apply to setter methods. We find that developers prefer to centralize data storage, but favor a much more decentralized and domain-independent approach to data retrieval.
- If a class defines a getter method, then there is an approx. 50% chance that this class also defines a setter method. The distribution of the ratio between getters and setter methods fits a normal distribution.
- In general, the number of getter methods exceeds the number of setter methods by 2 to 1 in any given system. However, there occur always a few classes in each system that contain more setter than getter methods. We find that 0.5%-10% of all classes in 90% of the systems define more setter than getter methods.
- All analyzed systems contain classes that only define getter and setter methods. We call these classes *pure data holders*. The general usage pattern for pure data holders follows a normal distribution with a mean value of approx. 8%.

The rest of this paper is organized as follows: in Section II we discuss our analysis method, in particular, we motivate the use of the *Lorenz curve* [16], Gini coefficient [15], and *decision frames* [17] for the analysis of software systems. We present our experimental method in Section III and proceed with an interpretation of our observations in Section IV. We conclude in Section V with a summary of our findings and sketch out some future work in this area.

II. Decision Frames

Skewed distributions with low mean values and large variances are common in many measured natural and man-made entities [18] and make interpretations using central tendency measures difficult if not impossible. Software metrics data is no exception [14], [19], [20]. Consider, for example, the distribution profile for getter methods in eclipse-3.2.2 as shown in Figure 1. This profile clearly does not follow a Gaussian distribution. In fact, 64.56% of the classes in eclipse-3.2.2 do not have getter methods at all. However, there is one class that defines 87 getter methods. It is the long tail of a given software metrics distribution that is most informative, as we find there the highest concentrations of features under investigation.

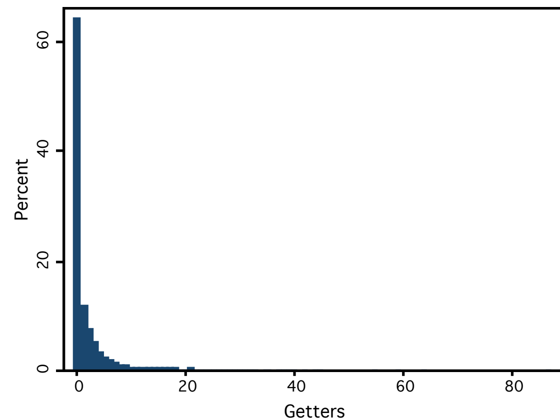


Figure 1. Positively skewed getter distribution in eclipse-3.2.2.

While the skewness cannot necessarily be attributed to a built-in bias [21], empirical evidence shows [14], [19], [20] that developers tend to prefer certain solution designs more than others. Moreover, in a recent study of human behavior, Souman et al. [21] argue that the effects causing a skewness in a distribution are due to an *accumulation of noise*. The variability induced by this noise does not lead to a Gaussian distribution, as the individual parameters causing the effects are multiplicative, hence preventing the formation of a normal distribution [18]. In the figurative sense, when

applied to software development this suggests that developer decisions have *multiplicative* effects too.

Consider, for example, the following scenario: Given an existing software system, we wish to incorporate some new functionality into it in order to address changing requirements. We have two options available:

- Add a new class hierarchy,
- Extend the existing classes.

Empirical evidence shows that developers choose the "extend" option slightly more often than the "add" option [20]. Although this preference is small, each time such a decision needs to be made the underlying bias causes an increased accumulation of functionality in a small subset of classes over time. As a consequence, we observe, for example, a skewed distribution profile for the number of methods in a class [14], [19]. In other words, even though developer decisions are influenced by multiple factors, the presence of a consistent bias in a subset of these factors has skewed distribution profiles as a natural outcome.

To cope with skewed metrics data distributions and effectively interpret them Vasa et al. [14] proposed a technique that uses the *Lorenz curve* [16] and the *Gini coefficient* [15]. Both yield expressive summary statistics that allow for an intuitive interpretation of the underlying data without the need to fit it to some known distributions.

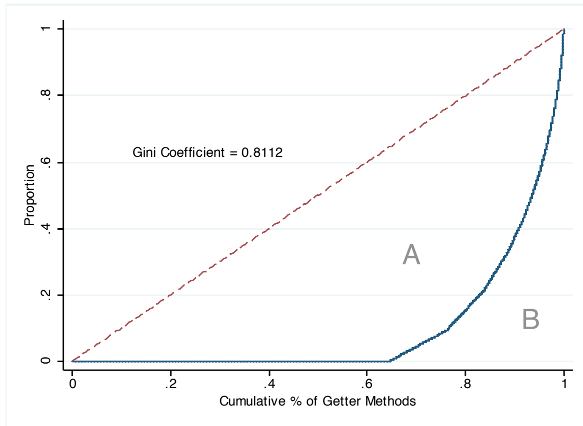


Figure 2. Lorenz curves for getter methods in eclipse-3.2.2.

In 1905 Lorenz [16] introduced a graphical representation to study the distribution of wealth in a population. Let w be the total income of the members of a population with an income of at most v and let W be the total income of the whole population. In the Lorenz curve, we plot w/W against the cumulative proportion of the population having an income of at most v. If the income happens to be equality distributed, then the Lorenz curve produces a straight diagonal line – *perfect equality*. However, an unequal distribution makes the Lorenz curve bend in the middle. This effect becomes stronger the higher the concentration of wealth is in a population. So, if $f(x)$ is a probability density functions and $F(x)$ a cumulative density function, then the Lorenz curve $L(F(x))$ is defined as:

$$L(F(x)) \quad = \quad \frac{\int_{-\infty}^{x} t\ f(t)\ dt}{\int_{-\infty}^{\infty} t\ f(t)\ dt} \qquad (1)$$

We can use the Lorenz curve to interpret (skewed) software metrics data. Consider, for example, Figure 2, which shows the Lorenz curve for the distribution of getter methods in eclipse-3.2.2. Getter methods in eclipse-3.2.2 are concentrated in a few classes. The Lorenz curve signifies this vividly. Only 35.44% of the classes in eclipse-3.2.2 actually define getter methods and the top 10% of the classes contain 60% of all the getter methods.

The Lorenz curve captures the nature of a distribution graphically. In order to assign this representation a numerical value, we can use the Gini coefficient that is defined as a ratio of the areas on the Lorenz curve diagram. If the area between the $45°$ line of perfect equality and the Lorenz curve is A, and the area under the Lorenz curve is B, then the Gini coefficient is $A/(A+B)$ [22]. More formally, if the Lorenz curve is $L(X)$, then

$$G \quad = \quad 1 - 2 \int_{0}^{1} \ L(X)\ dX \qquad (2)$$

In the case of the getter methods for eclipse-3.2.2, the Gini coefficient is 0.8112, a value that signals a high inequality in the distribution of getter methods in eclipse-3.2.2. But what does 0.8112 actually mean? In particular, is there an *expected value* for Gini coefficients that we can assume like the mean-value in a Gaussian distribution?

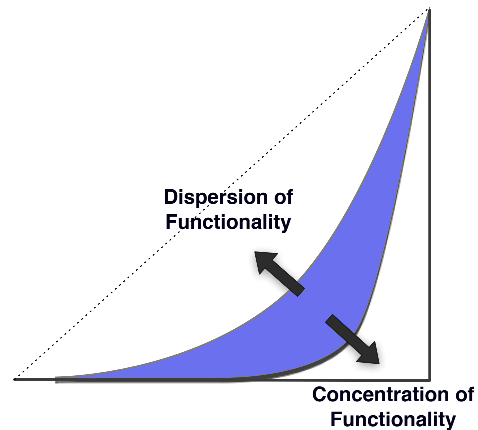


Figure 3. Feature decision frame.

The Gini coefficients for the same measure vary across systems [14]. For example, the Gini coefficients for getter methods in the 102 studied systems all assume values that fall into the interval $[0.596, 0.922]$. However, when we plot their corresponding Lorenz curves all in on the same surface, then a distinct pattern emerges (cf. Figure 3) – the *decision frame* of the measured feature. The term decision frame

refers to a principle formulated by Tverski and Kahneman [17] in which so-called "decision makers" proactively organize their solutions within strong boundaries prescribed by cultural environment and personal preferences. Changes in developer behavior result in a new decision frame. When using a more equitable solution strategy (*i.e.*, dispersion of functionality) the Gini coefficients will decrease and the corresponding Lorenz curves will move closer to the $45°$ perfect equality line. On the other hand, when developers apply more centralized designs (*i.e.*, concentration of functionality), then the Gini coefficients will grow accordingly and the Lorenz curves will bend further.

Based on our data set, we can empirically quantify the *width* of a decision frame and determine typical *min* and *max* values for each measure by using the *interquartile range* (IQR). Q_1 and Q_3, the lower and upper quartile, capture 50% of the data and yield a good estimate for the width of the decision frame. For example, we can establish the decision frame for getter methods by taking $Q_1 = 0.772$ and $Q_3 = 0.840$ with a median $Q_3 - Q_1 = 0.811$ resulting in decision frame interval $[0.772, 0.840]$ with width 0.068. We can, therefore, conclude that the getter method concentration in eclipse-3.2.2, denoted by the Gini coefficient 0.8112, fits perfectly the observed norm for the definition of getter methods in Java software.

The values outside the interval indicated by the lower and upper quartile, Q_1 and Q_3, do not necessarily signal problems, but rather specific, possibly domain-dependent, design decisions. Whether or not to trigger alarms in these cases depends on project-specific settings. We cannot predict the initial value of the Gini coefficients for software metrics data. However, once a Gini coefficient has been recorded for a particular measure, it moves little, typically less than 0.1 over the lifetime of a system, but always within the bounds of the associated decision frame [14].

III. METHODOLOGY

Java programs are translated into a machine-independent representation consisting of virtual machine instructions, called *bytecode* and embedded type information, called *metadata*. Both can be used to extract desired software metrics data.

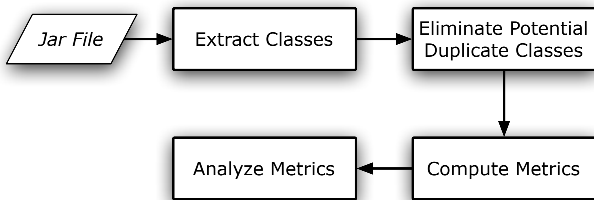


Figure 4. The metrics data mining process.

To capitalize on the information conveyed by bytecode and metadata we developed a data mining tool, called *JCFAnalyzer*, as part of a Masters project [23]. This tool takes a Java system, usually consisting of multiple jar-files, and distills the desired metrics data. All classes are counted as is. We do not attempt to collapse inner classes into their host classes. The output is a comma-separated text file that is subject to some post-processing in order to eliminate duplicates (cf. Figure 4).

Table I
THE UPDATED INPUT DATA SET.

Removed Systems	Added Systems
nekohtml-0.9.5	abdera-0.4.0-incubating
jasml-0.10	acegisecurity-1.0.7
cobertura-1.9	activemq-5.1.0
jparse-0.96	bcel-5.2
oscache-2.3-full	castor-1.3
quilt-0.6-a-5	cayenne-2.0.4
jgraph-5.12.1.0	cxf-2.2
fitjava-1.1	groovy1.6.1
picocontainer-1.3	iBATIS-2.3.4
jsXe-04_beta	maven-2.1.0
jFin_DateMath-R.1.0.0	mina-2.0.0-M4
quickserver-1.4.7	openejb-3.1
webmail-0.7.10	openjpa-1.2.1
displaytag-1.1	tapestry-5.1.0.2
freecs-1.2.20060130	tiles-2.1.2
javacc-3.2	turbine-2.3.3
	webwork2.2.7
	wicket-1.2.7

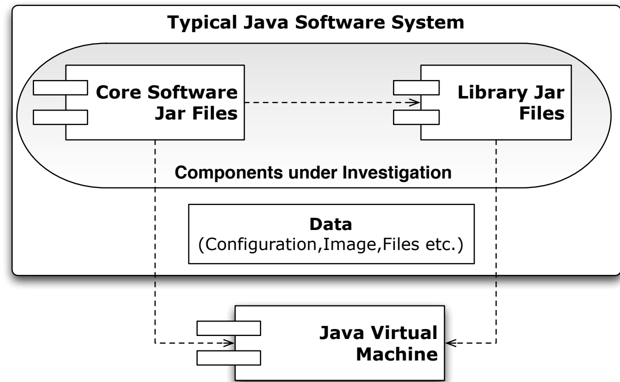


Figure 5. The scope of the data analysis.

A. Input Data Selection

Our reference data set for this study is the *Qualitas Corpus* [5]. The corpus contains 100 open-source Java systems. However, some of the systems are very small (*i.e.*, contain between 47 and 135 classes) and, due to the small resulting metrics data sets, their statistical significance with respect to studied properties is rather limited. For this reason, we decided to eliminate systems with less than 150 classes from the study. As a consequence, we were left with only 84 suitable candidate systems from the *Qualitas Corpus*.

Table II
THE COLLECTED FIELD AND METHOD MEASURES.

Field Measures		
Name	**Purpose**	**Relation**
Number of Attributes (NOA)	Counts all defined fields in a class	-
Number of Public Attributes (NOPubA)	Counts all public fields in a class	NOPubA $\subseteq$ NOA
Number of Protected Attributes (NOProA)	Counts all protected fields in a class	NOProA $\subseteq$ NOA
Number of Private Attributes (NOPriA)	Counts all private fields in a class	NOPriA $\subseteq$ NOA
Number of Package Attributes (NODefA)	Counts all fields with default or package visibility in a class	NODefA $\subseteq$ NOA

Method Measures		
Name	**Purpose**	**Relation**
Number of Methods (NOM)	Counts all defined member functions in a class	-
Number of Getters (NOG)	Counts all member functions in a class, whose name starts with the prefix “get”	NOG $\subseteq$ NOM
Number of Setters (NOS)	Counts all member functions in a class, whose name starts with the prefix “set”	NOS $\subseteq$ NOM
Number of Real Getters (NORG)	Counts all getter methods that access a field defined in the host class	NORG $\subseteq$ NOG
Number of Virtual Getters (NOVG)	Counts all getter methods that access a field defined outside in the host class	NOVG $\subseteq$ NOG
Number of Pure Getters (NOUG)	Counts all getter methods with 3-instruction sequence `aload_0`, `getfield`, and `[i|l|f|d|a]return`	NOUG $\subseteq$ NOG
Number of Real Setters (NORS)	Counts all setter methods that alter a field defined in the host class	NORS $\subseteq$ NOS
Number of Virtual Setters (NOVS)	Counts all setter methods that alter a field defined outside in the host class	NOVG $\subseteq$ NOS
Number of Pure Setters (NOUS)	Counts all setter methods with 4-instruction sequence `aload_0`, `aload_1`, `putfield`, and `return`	NOUS $\subseteq$ NOS
Number of Pure Getter-like Methods (NOGLM)	Counts all methods that act like pure getters (*i.e.*, have 3 instructions), but whose method names is not prefixed with “get”	NOGLM $\subseteq$ NOM
Number of Pure Setter-like Methods (NOSLM)	Counts all methods that act like pure setters (*i.e.*, have 4 instructions), but whose method names is not prefixed with “set”	NOSLM $\subseteq$ NOM

To replenish our input data set, we searched for additional open-source Java system and identified 18 systems that met our selection criterion. Table I provides a summary of the changes to the input data set.

The actual scope of our data analysis is illustrated in Figure 5. Each system may comprise of a core, additional support libraries, and data. We consider for our analysis only the core and support libraries. However, in order to avoid counting a particular classes twice, we remove all duplicates occurring in both the core and the support libraries from a system's data set prior analysis.

B. Measures

For the analysis of the usage patterns of getter and setter methods we collect 16 different software metrics split in *field measures* and *method measures* (cf. Table II). Both are required in order to properly ascertain the relationships between the fields defined in a class and their actual exposure by means of getter and setter methods. In particular, we seek to determine whether developers define getter and setter methods with distribution profiles similar to that of fields. However and without loss of generality, we only consider instance members, that is, we do not count static fields and methods.

The fields measures are *Number of Attributes*, the total number of fields in a class, and *Number of Public Attributes*, *Number of Protected Attributes*, *Number of Private Attributes*, and *Number of Package Attributes*, the number of occurrences of a field definition in a class with the corresponding visibility modifier or lack thereof.

The method metrics are divided into 4 categories: (i) *Number of Methods*, the total number of methods defined in a class, (ii) *Number of Getters*, the total number of getter methods in a class, and its semantic variants *Number of Real Getters*, *Number of Virtual Getters*, and *Number of Pure Getters* to record specific field access idioms, (iii) *Number of Setters*, the total number of setter methods in a class, and its semantic variants *Number of Real Setters*, *Number of Virtual Setters*, and *Number of Pure Setters* to count specific field update idioms, and (iv) *Number of Pure Getter-like Methods* and *Number of Pure Setter-like Methods*, methods that behave like “get”- and “set”-methods but do not follow the prescribed naming convention.

C. Threads to Validity

We rely solely on the information contained in the Java class files for our data analysis. But while class files offer a rich source of information about the actual software system, they cannot fully replace source code. We identify three particular issues:

- Inner classes are mapped by the Java compiler to separate class files. We do not attempt to recover the original structure, that is, we do not merge the metrics data of inner classes with that of the host

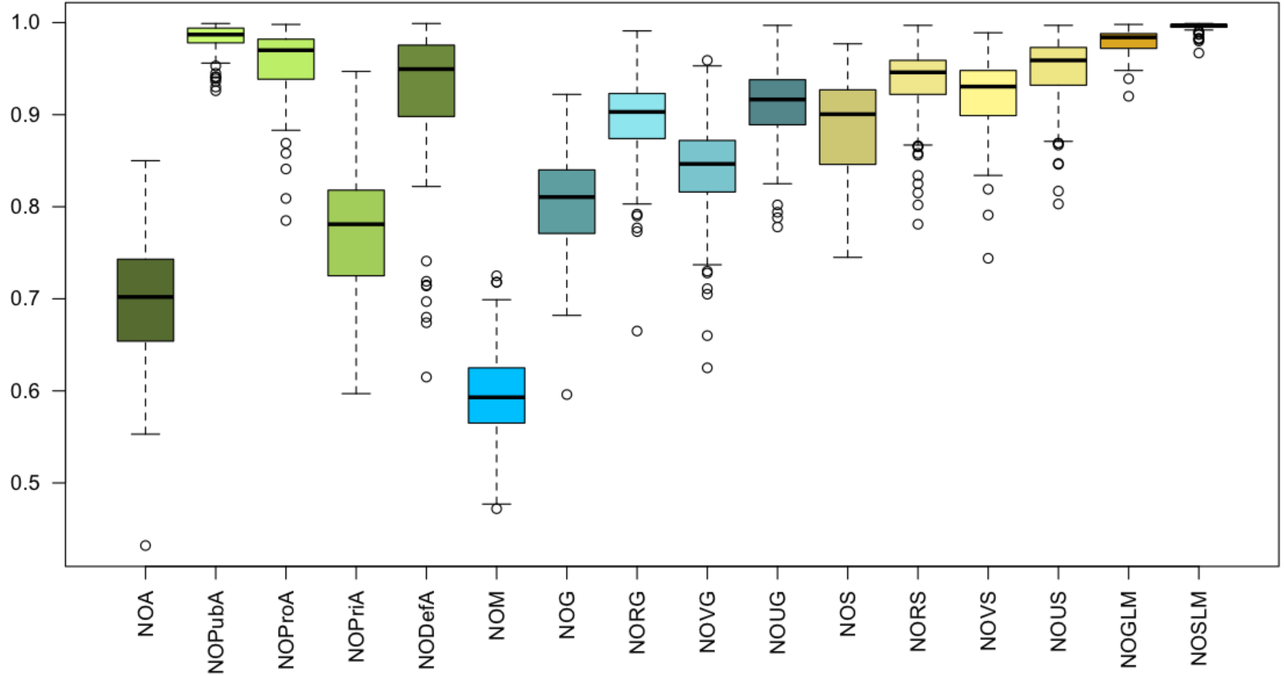


Figure 6. The box plots of all Gini coefficients for the 16 recorded measures.

classes. As a consequence, the actual population size to be considered increases and may result in a reduced concentration of features. In other words, we may observe lower Gini coefficients for a particular measure, as we treat inner classes as individuals rather than part of their host classes.

- The elimination of duplicates may influence the population size and give rise to lower Gini coefficients for particular measures. The actual effect is unknown. However, the duplicate elimination process, in general, only affects a small number of classes.
- We cannot detect getter- and setter-like method patterns other than *Number of Pure Getter-like Methods* and *Number of Pure Setter-like Methods* when inspecting bytecode alone. As a consequence, the analyzed systems may, in fact, contain additional getter- and setter-like methods (*i.e.*, methods with getter or setter semantics, but not adhering the the "get" and "set" naming pattern). Rectifying this problem requires analysis techniques beyond bytecode inspection.

IV. Discussion

We extracted our 16 defined measures and computed the corresponding Gini coefficients for the software metrics data for all 102 systems. In addition, we applied the Shapiro-Wilk test for normality [24] to a variety of promising measures in order to identify possible Gaussian distributions. With the help of the percentage points table of the W-test we were able to either accept or reject the *NULL*-hypothesis for normality for selected measures.

The range of the Gini coefficients of each measure is shown, in terms of box plots, in Figure 6. All measures reveal a preference to concentration of functionality with *Number of Methods* (NOM) exhibiting the least concentration and *Number of Pure Setter-like Methods* (NOSLM) almost achieving perfect inequality. However, we also observe that the key measures, *Number of Methods* (NOM), *Number of Attributes* (NOA), *Number of Getters* (NOG), and *Number of Setters* (NOS), all fall into well-defined intervals with narrow widths and assume mostly unique and non-overlapping values (cf. Table III).

Measure	Interval	Width
Number of Methods	$[0.565, 0.675]$	0.11
Number of Attributes	$[0.655, 0.743]$	0.088
Number of Getters	$[0.772, 0.840]$	0.068
Number of Setters	$[0.846, 0.927]$	0.081

Table III
Decision frames of key measures.

The ranges for *Number of Methods* and *Number of Attributes* echo that ones recorded by Vasa et al. [14], even though they show a slightly higher concentration of functionality. We credit this shift towards higher Gini coefficients

to the system selection and to the scope of this study. Vasa et al. [14] looked at 50 systems and their evolution resulting in more than 1,200 studied versions, whereas this study primarily focuses on a snapshot of the current state of 102 systems.

There are, however, some exceptions. The systems checkstyle-4.3 and jmoney-0.4.4 have comparatively low Gini coefficients for *Number of Methods*, that is, 0.472 and 0.477, respectively. Upon closer inspection, we discover that 66.4% of all classes in checkstyle-4.3 have between 2 and 7 methods and one class with 135 methods (ANTLR machine-generated code), whereas 67.6% of the classes in jmoney-0.4.4, built with JBuilder, have just 2 methods, which means that both systems enjoy a relatively equitable distribution of functionalities with respect to methods. On the other hand, trove-1.1b5 has a very low Gini coefficient for *Number of Attributes* (*i.e.*, 0.432). No class in trove-1.1b5 defines more than 4 fields and the majority of classes in trove-1.1b5 are type wrappers. Finally, we also recorded a relatively low Gini coefficient of 0.596 for *Number of Getters* in checkstyle-4.3. This is again a consequence of the architecture underpinning checkstyle-4.3, where 37.5% of the classes have no getter methods and 49% of the classes have 1 or 2 getter methods.

A. Field Distribution

Fields provide a facility to represent an object's state in object-oriented systems. In Java, we can assign each field a *visibility modifier*, which controls the degree of data encapsulation. A field can have either *public*, *protected*, *private*, or *package* visibility. But how do developers make use of these modifiers?

We inspected each of the 102 systems for the actual use of the visibility modifier. Not surprisingly, public fields are rare and occur very centralized in a few classes. The decision frame for *Number of Public Attributes* is $[0.978, 0.994]$ indicating a very high concentration of public fields in a few classes. In other words, developers appear to keep object state exposure through public fields at a minimum and when such an exposure is necessary it is confined to a few, specially designated, classes.

There are 13 systems with no public fields at all. However, we also found 3 systems (*i.e.*, openejb-3.1, proguard-3.6, and rssowl-1.2) in which more than 20% of the fields are public. These systems have Gini coefficients, 0.953, 0.925, and 0.935, respectively, which lie outside the decision frame for *Number of Public Attributes* indicating usual design choices. For example, proguard-3.6 is based extensively on the *Visitor pattern* with a large amount of duplicated code artifacts.

Almost all systems have fields with private, protected, or package visibility. Private fields show the least concentration followed by package and protected fields. Nevertheless, all field modifiers occur with varying frequencies and concentrations (see Figure 7). Moreover, even though private fields prevail overwhelmingly in most systems, in jmoney-0.4.4, for example only 15.16% are private fields, whereas 76.5% have package visibility. Only htmlunit-1.8 uses just private fields. Developers tend to concentrate public and protected fields more than fields with private and package visibility.

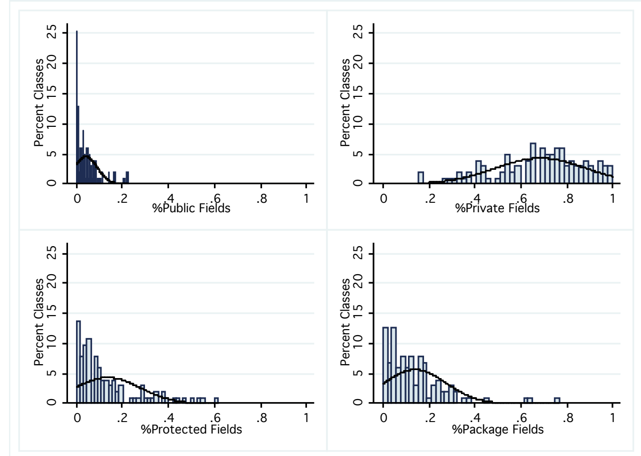


Figure 7. Field distribution by modifier.

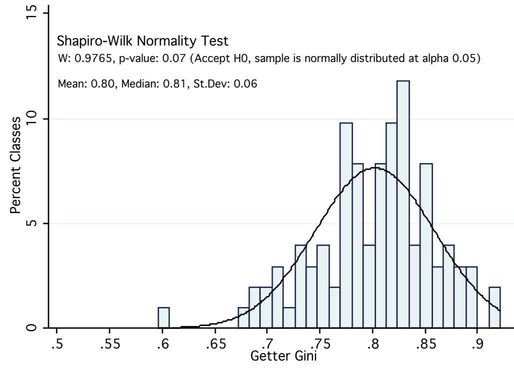
B. Getter and Setter Methods Distribution

Our main interest in this study is pertaining to the question of how developers use the property mechanism in Java and whether the problem domain has any impact on their use. Getter and setter methods are very unevenly distributed in software systems. Developers prefer to concentrate these methods in a limited number of classes. The corresponding distribution profiles are shown in Figures 8(a) and 8(b).

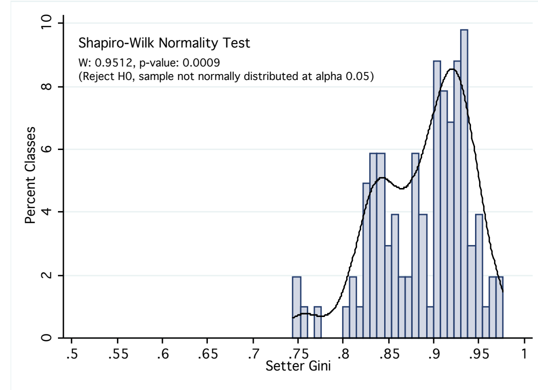
The Gini coefficients for getter methods range from 0.60 (*i.e.*, checkstyle-4.3) to 0.92 (*i.e.*, jruby-1.0.1) and follow a normal distribution with a mean of 0.80. The system with the least concentration of getter methods is poi-2.5.1 in which only 32.87% of the classes have no getter methods, whereas colt-1.2.0 is the system with highest concentration of getter methods (*i.e.*, 84.82% of classes have no getters).

For setter methods, the Gini coefficients range from 0.75 (*i.e.*, trove-1.1b5) to 0.97 (*i.e.*, junit-4.5). Unlike getter methods, the Gini coefficients for setter methods do not fit to a normal distribution. We observe two peaks around 0.83 and 0.93. The system with the least concentration of setter methods is ant-1.7.1 in which 53.41% of the classes have no setters. The system with the highest concentration of setter methods is junit-4.5, where 97.34% of the classes have no setter methods.

The property mechanism is not being used to circumvent the visibility modifiers, in particular, private. The concentration of getter and setter methods exceeds significantly that of private fields. There is no empirical evidence to support a claim that when developers define a private field they will also define a get- and set-method by default. We find that



(a) Getter method Gini distribution.



(b) Setter method Gini distribution.

Figure 8. The getter and setter method Gini coefficient distributions for the 102 investigated systems.

the opposite is true. Though get- and set-methods are used on a regular basis (all studied systems make use of them), developers consciously employ them in a manner consistent with domain and system requirements.

While the Gini coefficients for getter methods are normally distributed, the Gini coefficients for setter methods show a skewed distribution (cf. Figures 8(a) and 8(b)). Normal distributions arise when the underlying data is sufficiently large and the population is adequately diverse. A skewed distribution, on the other hand, emerges when a few factors contribute multiplicatively [18]. The getter method Gini coefficients are normally distributed, suggesting an independence from domain and solution design. The skewed nature of the setter method Gini coefficients, however, stipulates the opposite.

C. The Pairing of Getter and Setter Methods

If a class defines a getter method, then there is 34% chance that this class also defines a setter method, but the odds are not evenly distributed. However, extending the getter/setter ratio to $40/60$ and $60/40$, respectively, yields a 51% probability that a getter method is accompanied by some setter method with a profile that follows a normal distribution as shown in the corresponding *Q-Q plot* in Figure 9. This implies that the distribution and use of load and store operations in Java systems is not a function of the underlying problem being solved.

However, the dispersion of getter and setter methods is, in general, *inversely related* to the corresponding Gini coefficients (cf. Figure 10). Developers concentrate getter and setter methods in a few classes as much as possible. However, as the proportion of these methods grows the need to disperse them increases also and becomes unavoidable eventually. Though this seems natural, an important aspect of our finding is that even in cases where 30% of the methods are getters and setters, developers tend to centralize these methods in a few classes. The Gini coefficients for 30% of getter and setter methods assumes values typically greater than 0.7. This suggests that some level of bias towards model-separated design [25] is practiced.

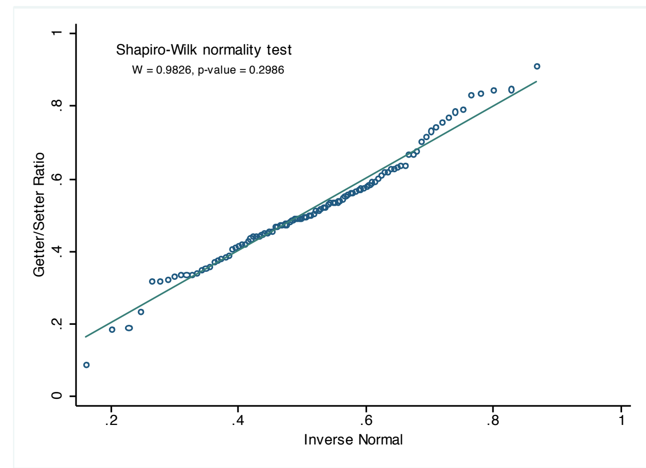


Figure 9. Getter and setter method ratio distribution.

D. Getter Dominance

Even though the number of getter methods exceeds the number of setter methods by 2 to 1, all analyzed systems contain classes with more setter than getter methods. In fact, between 0.5% and 10% of all classes in 90% of the systems define more setter than getter methods. However, we find a significant setter dominance only in ant-1.7.1, where 29.7% of all classes define more setter than getter methods. This consistent preference towards minimizing setter methods indicates that developers prefer to set up objects via constructors rather than exposing the object's state through setter methods.

E. Data Holder

All analyzed systems contain classes that only define getter and setter methods. We call these classes *pure data*

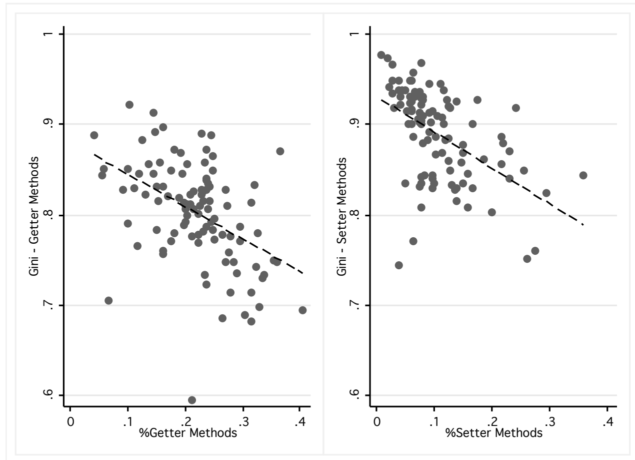


Figure 10. Percentage of getter and setter methods per Gini coefficient.

holders. The system with the least number of pure data holders is trove-1.1b5, in which only 0.18% of the classes fit the criterion. On the other hand, 31.89% of the classes in htmlunit-1.8 are pure data holders.

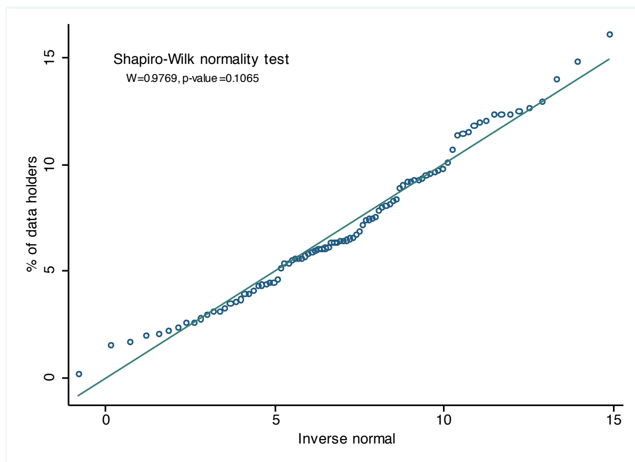


Figure 11. Distribution of data holders.

The amount of data holders present in a system is a function of design and desired functionality. Without additional information, we have to assume that the observed totals range well within what was targeted. What is remarkable about htmlunit-1.8, however, is the fact that it is also the one system in our study that has private fields only. Technically, htmlunit-1.8 is a library that allows Java programs to interact with Web-sites. The design model underpinning htmlunit-1.8 is *component inheritance* based, very similar to a typical graphical user interface framework. In htmlunit-1.8, HTML-pages are decomposed into hierarchies of elements, each having properties that can be retrieved via a sequence of get-methods. It is the library that uses set-methods to store information and users of the library use the get-methods to retrieve the information. This type architecture is inspired by *JavaBeans* [9], where the data is exposed and stored via a pair of getter and setter methods.

There are two other systems, webwork-2.2.7 and struts-1.2.9, that also contain a high number of pure data holders. We find 27.65% and 20.44% of the classes in webworks-2.2.7 and struts-1.2.9, respectively, are pure data holders. Again, we find pairs of getter and setter methods to retrieve and store information. It is, however, important to note that these cases are the exception. Moreover, the general pattern for pure data holders follows a normal distribution indicating domain and solution independence (cf. Figure 11).

F. Semantic Variants

Semantic variants of getter and setter methods capture specific behavioral idioms of the property mechanism in Java. A rather interesting fact in the 102 studied systems is that the concentration of both real getters and real setters exceed the that of virtual getters and virtual setters. Developers define get- and set-methods for fields defined in the same class in fewer instances than when they wish to expose a field from a different class. The underlying causes are yet unknown, but experience with the VCL of Delphi [6] indicates a particular design pattern: service super-classes provide the basic functionality that component subclasses require and which, in turn, expose the super-class fields through public get- and set-methods as needed.

All systems employ pure getter and setter methods, except emma-2.0.5312, which only uses pure getter methods. An aspect of more concern is that almost all systems also contain methods with get- and set-semantics, but are not designated as such. With the exception of webwork-2.2.7 all systems contain getter-like methods. Moreover, out of the 102 systems only 25 do not define setter-like methods. It must, however, be noted that the concentration of these methods is very high and the actual number of methods is very small. This suggests that these semantic variants offer little value to developers and if they are present in a system, then they occur most likely by accident.

V. Conclusion and Future Work

Getter and setter methods provide a controlled access to an object's state. Contrary to conventional belief, we find that these methods are neither commonplace nor "evil". Developers proactively select getter and setter methods in order to satisfy specific domain requirements. Certain domains require more getter and setter methods than others. We find that software systems based on the JavaBeans architecture [9], [10] are more likely to have many getter and setter methods. But this is just a result of the prescribed JavaBeans object model. In general, developers use getter and setter methods responsibly within clearly defined boundaries or *decision frames*.

The main weakness of Java with respect to property specification is its lack of an appropriate built-in language support. As a consequence, even though Java's "get" and "set" naming pattern can be used to develop software components (*i.e.*, JavaBeans), it is merely a guideline that can be easily abandoned or misused, as shown by the presence of getter- and setter-like methods in the investigated systems. However, additional work is required to properly ascertain whether built-in abstractions for property specification can assist developers in correctly applying the property mechanism in system design in the long run.

Developers naturally prefer to centralize data storage, but choose a much more decentralized and domain-independent approach to data retrieval. The main driving force for this bias appears to be aiming at maintainability, but a longitudinal study is required to support this hypothesis.

Such a longitudinal study can also reveal how the concentrations of getter and setter methods evolve over time. Is there a "comfort range" that developers target when defining getter and setter methods? We, therefore, plan as part of future work an analysis of getter and setter method allocation in evolving systems. In such a study we wish to determine, whether there exists some form of *cognitive preference* that drives developer decisions towards particular solution approaches.

REFERENCES

[1] P. Van Roy and S. Haridi, *Concepts, Techniques, and Models of Computer Programming*. The MIT Press, 2004.

[2] A. Goldberg and D. Robson, *Smalltalk-80: The Language*. Addison-Wesley, 1989.

[3] A. J. Riel, *Object-Oriented Design Heuristics*. Addison-Wesley, 1996.

[4] E. Tempero, "How Fields are Used in Java: An Empirical Study," in *Proceedings of 20th Australian Software Engineering Conference (ASWEC'09)*. Gold Coast, Queensland: IEEE Computer Society, 2009, pp. 91–100.

[5] Qualitas Research Group, "Qualitas Corpus release 20090202," http://www.cs.auckland.ac.nz/∼ewan/corpus, Feb. 2009.

[6] F. Bulback, *Programming Delphi Custom Components*. M&T Books, 1996.

[7] K. Arnold and J. Gosling, *The Java Programming Language*. Addison-Wesley, May 1996.

[8] M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, Apr. 1996.

[9] R. Englander, *Developing Java Beans*. O'Reilly, 1997.

[10] R. Monson-Haefel, *Enterprise JavaBeans*, 2nd ed. O'Reilly, 2000.

[11] N. E. Fenton and S. L. Pfleeger, *Software Metrics: A Rigorous & Practical Approach*, 2nd ed. Thomson Publishing, 1996.

[12] N. E. Fenton and M. Neil, "A critique of software defect prediction models," *IEEE Transactions on Software Engineering*, vol. 25, no. 5, pp. 675–689, 1999.

[13] B. A. Kitchenham, "An evaluation of software structure metrics," in *Proceedings of the 12th International Computer Software and Application Conference (COMPSAC 1988)*. IEEE Computer Society Press, 1988, pp. 369–376.

[14] R. Vasa, M. Lumpe, P. Branch, and O. Nierstrasz, "Comparative Analysis of Evolving Software Systems Using the Gini Coefficient," in *Proceedings of 25th IEEE International Conference on Software Maintenance (ICSM '09)*. Edmonton, Alberta: IEEE Computer Society, Sep. 2009, pp. 179–188.

[15] C. Gini, "Measurement of Inequality of Incomes," *The Economic Journal*, vol. 31, no. 121, pp. 124–126, Mar. 1921.

[16] M. O. Lorenz, "Methods of Measuring the Concentration of Wealth," *Publications of the American Statistical Association*, vol. 9, no. 70, pp. 209–219, Jun. 1905.

[17] A. Tversky and D. Kahneman, "The Framing of Decisions and the Psychology of Choice," *Science*, vol. 211, no. 4481, pp. 453–458, Jan. 1981.

[18] E. Limpert, S. W. A., and M. Abbt, "Log-normal Distributions across the Sciences: Keys and Clues," *BioScience*, vol. 51, no. 5, pp. 341–352, May 2001.

[19] R. Vasa, M. Lumpe, and J.-G. Schneider, "Patterns of Component Evolution," in *Proceedings of the 6th International Symposium on Software Composition (SC 2007)*, ser. LNCS 4829, M. Lumpe and W. Vanderperren, Eds. Braga, Portugal: Springer, Mar. 2007, pp. 235–251.

[20] R. Vasa, J.-G. Schneider, and O. Nierstrasz, "The Inevitable Stability of Software Change," in *Proceedings of 23rd IEEE International Conference on Software Maintenance (ICSM '07)*. Paris, France: IEEE Computer Society, Oct. 2007, pp. 4–13.

[21] J. Souman, I. Frissen, M. N. Sreenivasa, and M. O. Ernst, "Walking Straight into Circles," *Current Biology*, vol. 19, no. 18, pp. 1538–1542, Sep. 2009.

[22] K. Xu, "How Has the Literature on Gini's Index Evolved in the Past 80 Years?" Dec. 2004, Department of Economics, Dalhouse University, Halifax, Nova Scotia.

[23] S. Mahmud, "Frequency Analysis of Java Byte Code," Master's thesis, Swinburne University of Technology, 2010, in preparation.

[24] S. S. Shapiro and M. B. Wilk, "An analysis of variance test for normality (complete samples)," *Biometrica*, vol. 52, no. 3/4, pp. 591–611, 1965.

[25] E. Evans, *Domain-Driven Design – Tackling Complexity in the Heart of Software*. Addison-Wesley, 2004.