A Methodology to Evaluate the Maintainability of Enterprise Application Integration Frameworks

Rafael Z. Frantz

Department of Exact Sciences and Engineering,
Unijuí University,
Ijuí, RS, Brazil
E-mail: rzfrantz@unijui.edu.br

Rafael Corchuelo

Department of Language and Systems,
University of Seville,
Seville, Spain
E-mail: corchu@us.es

Fabricia Roos-Frantz

Department of Exact Sciences and Engineering,
Unijuí University,
Ijuí, RS, Brazil
E-mail: frfrantz@unijui.edu.br

Abstract: Consulting companies that specialise in Enterprise Application Integration commonly require adapting existing frameworks to specific domains. Currently, there are many such frameworks available, most of which provide a materialisation of the well-known catalogue of patterns that was devised by Hohpe and Woolf. The decision regarding which framework must be used is critical since adaptation costs are not negligible. In this article, we report on a methodology that helps practitioners make a decision regarding which framework should be selected. To the best of our knowledge, there is not a previous methodology in the literature. Its salient features are that we have assembled a catalogue of measures regarding which there is a consensus in the literature that they are clearly aligned with the effort required to maintain a piece of software and we propose a statistically-sound method to produce a rank. We illustrate our proposal with an industrial case study that we have performed using five open-source frameworks.

Keywords: Enterprise Application Integration; Software Maintainability; Adaptive Maintenance.

Biographical notes: Dr. Rafael Z. Frantz is an Associate Professor who is with the Department of Exact Sciences and Engineering of the Unijuí University, Brazil. He was awarded a PhD degree in Software Engineering by the University of Seville, Spain. His current research interests focus on the integration of enterprise applications and search-based software engineering.

Dr. Rafael Corchuelo is a Reader in Computer Science who is with the Department of Computer Languages and Systems of the University of Seville, Spain. He received his PhD degree from this University, and he leads its Research Group on Distributed Systems since 1997; his current research interests focus on the integration of web data islands; previously, he worked on multi-party interaction and fairness issues.
Dr. Fabricia Roos-Frantz is an Associate Professor who is with the Department of Exact Sciences and Engineering of the Unijuí University, Brazil. She received her PhD in Software Engineering from the University of Seville, Spain. Her current research interests include software product lines and search-based software engineering. She served as a reviewer for the SPLC'09 and SPLC'10.

1 Introduction

Typical companies run software ecosystems (Messerschmitt and Szyperski, 2003) that consist of many applications that support their business activities. Frequently, new business processes have to be supported by two or more applications, and the current business processes may need to be optimised, which requires interaction with other applications. However, it is common that these applications were not designed with integration concerns in mind, i.e., they do not provide a programming interface. As a result, the interaction is not always a trivial task, and has to be carried out in most cases by means of the resources that belong to the applications, such as their databases, data files, messaging queues, and user interfaces. Recurrent challenges are to make the applications inter-operate with each other to keep their data synchronised, offer new data views, or to create new functionalities (Hohpe and Woolf, 2003).

Enterprise Application Integration (EAI) is a broad research field in software engineering that focuses on providing methodologies and tools to integrate the many heterogeneous applications of typical companies' software ecosystems. It aims to keep a number of applications' data in synchrony or to develop new functionality on top of them, in a way that applications do not have to be changed and are possibly minimally or not affected by the integration solution (Hohpe and Woolf, 2003). From the application viewpoint, they are not aware that they take part of an integration solution.

The ultimate goal in the field of EAI is to provide companies with technologies, tools, and methods that help cut integration costs off (Chalmeta and Pazos, 2015; Hitt et al., 2002; He and Xu, 2014). In this article, we focus on integration frameworks. Such frameworks are general-purpose tools that are intended to facilitate developing integration solutions in a general context. There are cases in which a company decides to adapt one of them to a specific domain (Chen and Huang, 2009), e.g., e-commerce (RosettaNet, 2011), health (HL7, 2011) finances (Swift, 2011), or insurance (HIPAA, 2011). In such cases, the company first needs to select amongst the existing frameworks the ones that provide the functionalities or the connectors that they need and then make a decision regarding which one should be adapted to meet the specific requirements of a specific domain.

It is not surprising that how a framework was designed and implemented has an impact on its maintenance costs (Epping and Lott, 1994; Jorgensen, 1995; Bergin and Keating, 2003; Schneidewind, 1987). In both design and implementation, software engineers need to pay attention to readability, understandability, and complexity, since they are related to maintainability. The models and source code must be easy to read and understand, because it is very common that the people who work on them are not involved in their maintenance. The complexity of the algorithms should be kept low, not only for performance reasons, but because it makes it easier for a software engineer to follow their execution flows and debug them. Thus, to reduce the costs involved in the adaptation of a framework to a specific context, it is very important that the framework was designed taking into account issues that have a negative impact on maintenance.

Amongst the most recent and important open-source integration frameworks available for companies to design and implement integration solutions, there are Camel (Ibsen and Anstey, 2010), Spring Integration (Fisher et al., 2010), Mule (Dossot and D'Emic, 2009), and Guaraná (Frantz and Corchuelo, 2012). They are based on the catalogue of integration patterns documented by Hohpe and Woolf (2003), which have turned into a cookbook for software engineers to design and implement integration solutions. Furthermore, the core implementation of these frameworks is equivalent in functionality and they support the core concepts of pipes, filters, and resource adapters. Thus, in this article we selected them to demonstrate our proposal.

Given two different frameworks, the only totally accurate means to determine which one is more maintainable and adaptable is to use them in a number of projects in which software engineers with very similar skills are asked to maintain and adapt them for a particular purpose. Unfortunately, that does not make sense in an industrial environment because of the costs involved. This has motivated many researchers to devise individual measures that are correlated to the effort required to maintain and adapt a piece of software (McCabe, 1976; Li and Henry, 1993; Chidamber and Kemerer, 1994; Henderson-Sellers, 1996; Briand et al., 1998; Sheldon et al., 2002; Martin, 2002; Bocco et al., 2005; Mouchawrab et al., 2005; Lanza and Marinescu, 2006; Lajios, 2009; Herraiz et al., 2009; Risi et al., 2013). Many of them have been validated in real-world projects (Lanza and Marinescu, 2006; Tempero et al., 2008; Balmas et al., 2009; Burger and Hummel, 2012; Mordal-Manet et al., 2013). Thus, these measures can be effectively used in practice to analyse the maintainability and adaptability of integration frameworks.

However, throughout the years, several measures were developed and proposed separately, each one aiming at analysing only a single and very concrete aspect of the software system, such as the complexity of algorithms, the number of attributes in classes, the number of methods that share common attributes in a class, the depth of inheritance trees, etc. Thus it is not easy to use these measures in order to analyse the maintainability of integration frameworks. It is not easy to decide on which measures must be used and the current lack of a methodology may lead to a wrong interpretation of the collected data and then to an incorrect measurement of the maintainability.

In this article, we have managed to assemble a catalogue of twenty-five different measures in the literature that can help forecast how maintainable an integration framework is, and we also propose a statistically sound method to analyse the results. We have classified these measures into four categories based on the model proposed by Lanza and Marinescu (2006), namely: size, coupling, complexity, and inheritance. We also report on an analysis regarding Camel, Spring Integration, Mule, and Guaraná that was carried out in an industrial context. To the best of our knowledge, no such a methodology or study has been presented previously in the context of enterprise application integration frameworks.

An empirical study to analyse maintainability in web-based systems was conducted by Chae et al. (2007), which have found that the same measures used to predict the maintainability of conventional systems cannot be used to predict the maintenance effort on web-based systems. This is an interesting work since it suggests measures may have a strong influence of the kind of software being analysed. Thwin and Quah (2003) have studied the application of neural networks to estimate the maintainability of a software system. In this piece of work, they use nine object-oriented measures to predict the maintainability effort by estimating the number of lines changed per class. Our proposal does not focus on the number of lines changed, instead we have selected twenty-five measures from the literature and classified them into four categories that allow us to analyse how maintainable an integration framework is, namely: inheritance, complexity, coupling, and size. Furthermore, we propose a statistically sound method to analyse the results obtained directly from the application of these measures. Briand et al. (1999) have also studied the maintainability of software systems, but focusing on a single category of measure: coupling. They surveyed the literature on coupling measures and then proposed a unified framework that can be used to measure coupling in software systems. Their work is complementary to ours in that we share common coupling measures.

The prediction of maintainability effort at the design stage was studied by Nair et al. (2010). In their work the authors select a set of measures that can be used to analyse the impact of the design on the maintainability of a software system and then the authors claim that their approach can be used to produce cost effective software. The structural complexity of UML class diagram as means to predict the maintainability of object-oriented information systems was studied by Genero et al. (2001). In that work the authors present a set of measures and demonstrate through experimentation that these measures contribute to predict UML class maintainability. Heitlager et al. (2007) have analysed the use of the Maintainability Index (Oman and Hagemeister, 1994; Coleman et al., 1994) in real-world projects to estimate software maintainability and have found that it lacks information on which properties may have negatively influenced the maintainability of a software system, so that this is difficult to improve maintenance. Then, Heitlager et al. (2007) propose a practical model to estimate the maintainability by using a set of well-chosen source-code measures that are mapped onto the sub-characteristics of maintainability of ISO 9126 following pragmatic mapping and ranking guidelines. Antonellis et al. (2007) proposed a methodology that uses data mining to evaluate the maintainability of a software system according to the ISO 9126 quality model. They collect data for nine maintainability measures and store them in a relational database for further analysis and evaluation. This methodology focuses on the collection and presentation of values for the measures, leaving the analysis and conclusions to software engineers. Kumar et al. (2015) proposed a methodology to analyse the maintainability of software systems by using a non-linear model. Their methodology uses eleven object-oriented measures to collect maintainability data. From the analysis of the related work, it is possible to notice that all of them have their foundations on object-oriented measures, some of them only collect the vales for each measure leaving the analysis for software engineers, and none of them have analysed enterprise application integration frameworks. Our proposal takes into account three times more maintenance measures; we have surveyed the current literature on this topic, and we have carefully selected a subset of measures regarding which there is a clear consensus that they are clearly aligned with the effort required to maintain a piece of software; furthermore, having such a large collection of measures allows to provide as a complete view of a proposal as possible. Furthermore, we propose to use a statistically-sound method to compare them all; this method allows to compare several proposals at a time and keep the error rate under control; the resulting rankings then reflect differences that are statistically significant at a given significance level.

The rest of the article is organised as follows: Section 2 introduces our methodology; Section 3 reports on our application to four well-known frameworks; finally, Section 4 concludes the article.

2 Methodology

In this section we introduce our methodology. We first report on the maintainability measures that we have selected and constitute the foundation of our proposal and then on the statistical methodology to analyse them.

2.1 Size Measures

The size of a framework is influenced by the number of packages, classes, interfaces, attributes, methods, and their parameters. The measures in this group allow to understand how big a framework is.

NOP: Number of packages that contain at least one class or interface. This measure can be used as an indicator of how much effort it is required to understand how packages are organised; note that this provides the overall picture of the design of a framework (Dong and Godfrey, 2009). The greater this value, the more effort shall be required.

NOC: Number of classes. This measure and the following one can be used as indicators of how much effort shall be required to understand the source code of a framework (Lanza and Marinescu, 2006). The grater this value, the more difficult it is to understand a framework.

NOI: Number of interfaces. It is commonly agreed that the larger the number of interfaces, the easier to a adapt a framework (Lanza and Marinescu, 2006).

LOC: Number of lines of code, excluding blank lines and comments (Lanza and Marinescu, 2006). In general, the greater this value, the more effort shall be required to maintain a framework.

NOM: Number of methods in classes and interfaces. This measure can be used as an indicator for the potential reuse of a class. According to Lorenz and Kidd (1994), and Chidamber and Kemerer (1994), a large number of methods may indicate that a class is likely to be very application specific, which hinders its reusability.

NPM: Number of parameters per method. This measure can be used as an indicator of how complex it is to understand and use a method. According to Henderson-Sellers (1996), the number of parameters should not exceed five. If it does, the author suggests that a new type must be designed to wrap the parameters into a unique object. The greater this value, the more difficult it is to understand a method.

MLC: Number of lines in methods, excluding blank lines and comments. According to Henderson-Sellers (1996), this value should not exceed fifty. If it does, the author suggests to split this method into other methods to improve readability and maintainability. The greater this value, the more difficult it is to understand and maintain a method.

NSM: Number of static methods. This measure can be used as an indicator of how well implemented a piece of code is (Lanza and Marinescu, 2006). The greater this value, the more likely that the code tends to be based on the classical procedural paradigm and not on the object-oriented paradigm.

NSA: Number of static attributes. This measure can be used as an indicator of how difficult it is to reason about the state of a framework when testing (Lanza and Marinescu, 2006). The greater this value, the more difficult testing.

NAT: Number of attributes. This measure can be used as an indicator of how complex it is to understand a class (Lanza and Marinescu, 2006). The greater this value, the more difficult it is to understand the state of its objects.

2.2 Coupling Measures

In the object-oriented paradigm, an important characteristic is the encapsulation of data and the collaboration of objects to perform system functionalities. The measures in this group give an indication of how coupled the classes of a framework are.

LCM: Lack of cohesion of methods. In this context, cohesion refers to the number of methods that share common attributes in a class. It is computed with the Henderson-Sellers LCOM* method (Henderson-Sellers, 1996). A low value indicates a cohesive class; contrarily, a value that is close to one indicates lack of cohesion and suggests that the class might better be split into two or more classes because there can be methods that are likely not to belong to that class.

AFC: Afferent coupling. This measure is defined as the number of classes outside a package that depend on one or more classes inside that package. The greater this value, the more complex maintenance becomes because there are more dependencies between classes (Martin, 2002; Offutt et al., 2008; Yu, 2008). Furthermore, larger values of afferent coupling can be used as an indicator that the package is critical for the framework and then maintenance in this package must be performed carefully not to introduce problems in the dependent classes.

EFC: Efferent coupling. This measure is defined as the number of classes inside a package that depend on one or more classes outside the package. The greater this value, the more likely that maintenance shall have an impact on a package (Martin, 2002; Offutt et al., 2008; Yu, 2008).

FAN: Number of called classes. This measure can be used as an indicator of how dispersed method calls are in the classes of a framework (Lorenz and Kidd, 1994). The greater this value, the more complex a method call is because every call is supposed to involve other classes to be completed.

LAA: Locality of attribute accesses. This measure can be used as an indicator of how dependent the methods of a class can be regarding the attributes of other classes (Lanza and Marinescu, 2006). The greater this value, the more a method of a class uses attributes from other classes. The highest value is 1 and represent 100% dependent.

CDP: Coupling dispersion. This measure can be used as an indicator of bad method design, since a method may be executing more than one thing and then can be split reducing its coupling (Lanza and Marinescu, 2006). The greater this value, the more likely that there is an improper distribution of functionality amongst the methods of a framework. The highest value is 1 and represent 100% dispersion.

CIT: Coupling intensity. This measure can be used as an indicator of how dependent a method is, since it measures the number of distinct methods that it calls (Lanza and Marinescu, 2006). The greater this value, the more likely there is an excessive coupling amongst the methods of a framework.

2.3 Complexity Measures

The notion of complexity is important in frameworks, chiefly if they have to be maintained. The measures in this group allow to understand how complex a framework is.

ABS: Degree of abstractness of a framework. This measure can be used as an indicator of how customisable a framework is (Martin, 2002). The greater this value, the easier to customise the framework. The highest value is 1 and represent 100% abstract.

WMC: Weighted sum of the McCabe cyclomatic complexity (McCabe, 1976) for all methods in a class. This measure can be used as an indicator of how difficult understanding and then modifying the methods of a class shall be (Chidamber and Kemerer, 1994). The greater this value, the more effort is expected to maintain a class.

MCC: The McCabe cyclomatic complexity. This measure can be used as an indicator of how complex a method is. According to McCabe (1976), this value should not exceed ten. The greater this value, the more difficult it is to maintain a piece of code.

WOC: Weight of class. This measure indicates the ratio of accessor methods regarding other methods that provide services (Marinescu, 2002). The greater this value, the more class interfaces consists of accessor methods, which indicates that classes are not too complex. The highest value is 1 and represent 100% of accessor methods in a class.

DBM: Depth of nested blocks in a method. This measure can be used as an indicator of how expensive debugging a piece of code can be. According to Henderson-Sellers (1996), this value should not exceed five. If it does, the author suggests that the method should be split into several methods. The greater this value, the more complex e method is.

2.4 Inheritance Measures

Inheriting functionalities amongst classes is a well-known characteristic of the object-oriented paradigm. The measures in this group report on how much and how well inheritance is used.

DIT: Depth of Inheritance Tree. Inheritance is a mechanism that increases core reuse (Alkadi and Alkadi, 2003). This measure can be used as an indicator of how complicated maintaining a class can be. The greater this value, the more difficult to maintain a framework.

NOH: Number of immediate children classes of a class. This measure can be used as an indicator of the potential impact that a class may have in a framework if it is modified (Chidamber and Kemerer, 1994). The greater this value, the greater the chances that the abstraction defined by the parent class is poorly designed.

NRM: Number of overriden methods. This measure can be used to indicate how adaptable a class is with respect to its ancestors (Lorenz and Kidd, 1994). The greater this value, the more likely that the inheritance mechanism is being used to adapt a class instead of just providing additional services to the parent class.

2.5 Statistical Analysis

Computing the values of a number of measures regarding a number of proposals a comparing them using their average value may easily lead to wrong conclusions. The problem is that the average value of a measure may be highly influenced by its distribution of values. For instance, a uniformly distributed measure and a highly skewed measure may have the same average, but they behave quite differently in practice. Analysing the standard deviation in addition to the average may help, but then the problem is how to compare two different indicators at the same time. Even if we could compare them both, a problem would remain: we need to make sure that the empirical data supports the hypothesis that the differences are statistically significant, that is, that they are due to intrinsic features of the proposals that are being compared and due to the implicit random effect that exist in any real-world experimentation. Traditionally, the previous question has been addressed by using parametric tests (Sheskin, 2012). Unfortunately, they assume that the values of the measures are distributed normally; many of them also require them to be homoscedastic, i.e., their variances to be equal; furthermore, many of them can only deal with two proposals at a time and using them with more proposals results in uncontrollable error rates. Although these limitations have not usually hindered the applicability of parametric tests in areas such as medicine, biology, demographics, and the like, it hinders their applicability to compare algorithmic proposals. In such areas, measures are not likely at all to be distributed normally and they are even less likely to have equal variances because they do not represent natural data, they are completely artificial; furthermore, it is very common that several proposals to solve the same problem exists and that they all have to be compared to each other. Very recently, this has motivated many authors in the field of statistics to work on so-called non-parametric tests (Demšar, 2006), which work on the empirical ranks of the measures instead of their values; this makes this kind of tests independent from the distribution of values of the measures analysed, more resilient to outliers, and, thus, less prone to draw wrong conclusions regarding artificial measures.

Figure 1 illustrates the steps of the methodology that we have devised, and we describe them below:

1. Compute Measures. Calculate the measures for every proposal and collect them. There are several software tools that can be used to calculate the maintainability measures in an automated fashion for an integration framework. Frequently, these tools take as input packages of source code so that they can be analysed and the values calculated for every measure. In our research, we have used two different and complementary free software tools in order to compute the twenty-five measures we have selected from the literature, namely: Metrics 1.3.6[1] (Metrics, 2015) and iPlasma 6.1[2] (Lanza and Marinescu, 2006).

2. Compute Rank. Compute the empirical rank of each proposal regarding the measure being analysed. When the values of the measures range in intervals that are not homogeneous, it is generally a good idea to normalise them to interval $[0, 1]$ since this

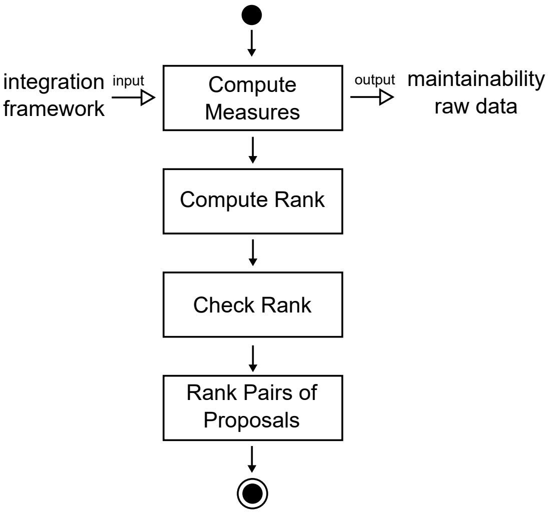

Figure 1 Overview of the proposed methodology.

will make comparisons more intuitive; otherwise, the difference in scale may easily lead to misinterpretations.

3. Check the Rank. Check if the differences in the empirical ranks can be considered statistically significant or not at a given significance level α. (Typically α is set to 0.05, that is 95% confidence.) This check can be easily implemented using Iman-Davenport's test. If there are not any significant differences, then the conclusion is that the data that we have collected does not provide any evidences that the proposals that we are comparing behave differently regarding the measure being studied; contrarily, if the differences are significant, it then makes sense to compare the proposals side by side to induce a statistical ranking.

4. Rank pairs. The statistical ranking can be easily implemented using Bergmann-Hommel's test. This test compares every pair of proposals regarding a measure, but keeps the error rate of the comparisons under strict control. Unfortunately, this test does not necessarily result in a total pre-order. Generally speaking, such situations occur when there is a minimal sequence of proposals $p_1, p_2, \ldots, p_n$ such that the test does not find enough evidence to conclude that p_i behaves differently from p_{i+1} for every $i = 1n - 1$, but it finds enough evidence to conclude that p_1 behaves differently from p_n. Our proposal to transform such chains into total pre-orders is to break them assuming that p_j does not behave like p_{j+1}, where p_j and p_{j+1} $(1 \leq j < n)$ denote the pair of proposals for which the test returns the smallest p-value above the significance level α; in other words, we suggest selecting the couple of proposals for which the experimental data provides more evidence that they behave differently. There is obviously a chance to make a mistake, but it is the only way to transform the results of Wilcoxon's Rank-Sum test into a total pre-order. Note that the decision might be taken arbitrarily at any other point in the chain and the results would be the same: there is only a chance to make a mistake at the point where the chain is arbitrarily broken.

In the literature on statistics, there are many recent non-parametric tests available. The decisions that we have made regarding selecting specific tests for our methodology build on the experimental studies that were carried out by Demšar (2006) and García et al. (2010).

3 Sample Application

In this section, we apply our methodology to evaluate the maintainability regarding the core implementation of Camel, Spring Integration, Mule, and Guaraná.

3.1 Preliminaries

In this sample application study we focus only the core implementation of the selected integration frameworks, i.e., we do not take into account the code required to implement the adapters that are required to interact with the applications being integrated. We do not consider this code because it is peripheral and, more often than not, comes from other open-source projects that are maintained separately.

Camel is a Java-based software tool that aims to provide an integration framework with a fluent API (Fowler, 2010) to support the design and implementation of Enterprise Application Integration solutions based on integration patterns. It was designed to be used by means of a Java- or a Scala-based domain-specific language, or by means of declarative XML Spring-based configuration files. The Java-based domain-specific language approach is the most popular in the Camel community. Camel is an open source tool that is hosted by the Apache Software Foundation. FuseSource is the company that provides products based on Camel, which includes a commercial version of Camel, a web-based graphical editor, and an Eclipse-based IDE with a graphical editor.

Spring Integration is also a Java-based software tool built on top of the Spring Framework container. It aims to extend this framework to support the design and implementation of Enterprise Application Integration solutions. Following the philosophy of Spring Framework, Spring Integration promotes the use of XML Spring-based files to configure integration solutions, although it is also possible to use Spring Integration as a command-query API (Fowler, 2010). Spring Integration is an open source tool that includes an Eclipse-based IDE with a graphical editor. The tool is supported by SpringSource, a division of company VMware Inc. VMware does not commercialise an enterprise version of Spring Integration, instead they use individual Spring Integration components in their commercial tools, such as vFabric RabbitMQ and vCenter Orchestrator.

Mule is a Java-based software tool whose architecture is inspired by the concept of enterprise service bus. It aims to support the design and implementation of Enterprise Application Integration solutions based on integration patterns. It was designed to be used by means of a command-query API (Fowler, 2010) or declarative XML Spring-based configuration files. The latter seems to be the most popular and recommended approach by the Mule community. Mule is open source and provides a community version that includes an Eclipse-based IDE with a graphical editor. A commercial enterprise version is also supported MuleSoft Inc.

Guaraná is an Enterprise Service Bus conceived to support software engineers in the design, implementation, and execution of application integration solutions. For design, Guaraná provides a Domain-Specific Language (DSL), which allows software engineers to keep their focus on the problem by using a graphical and very intuitive modelling

	Measure	Camel				Spring Integration				Mule				Guaraná			
		Total	Mean	Dev.	Max	Total	Mean	Dev.	Max	Total	Mean	Dev.	Max	Total	Mean	Dev.	Max
Size	NOP	54	-	-	-	32	-	-	-	124	-	-	-	18	-	-	-
	NOC	730	13.52	19.55	96	269	8.41	10.52	58	733	5.91	7.40	51	79	4.39	3.09	11
	NOI	140	2.59	9.07	58	40	1.25	1.84	9	209	1.69	3.28	18	9	0.50	0.76	2
	LOC	62,439	-	-	-	14,929	-	-	-	67,090	-	-	-	2,878	-	-	-
	NOM	7,015	9.61	15.36	192	1,431	5.32	5.60	39	5,158	7.04	10.23	129	369	4.67	4.61	24
	NPM	-	0.93	1.05	11	-	1.13	0.94	9	-	0.92	1.07	19	-	1.20	1.04	4
	MLC	34,839	4.52	8.15	141	8,264	5.65	9.59	110	35,989	6.16	10.99	180	1,748	4.72	6.43	54
	NSM	709	0.97	4.95	74	31	0.12	0.73	8	686	0.94	9.41	244	1	0.01	0.11	1
	NSA	291	0.40	1.07	16	109	0.41	1.52	13	669	0.91	3.66	81	30	0.38	1.33	10
	NAT	1803	2.47	4.17	62	474	1.76	2.51	16	1417	1.93	3.21	31	87	1.10	2.14	12
Coupling	LCM	-	0.29	0.35	1	-	0.22	0.33	0.94	-	0.23	0.34	1.33	-	0.14	0.27	0.91
	AFC	-	30.63	89.34	542	-	12.69	26.65	146	-	22.90	56.25	493	-	6.94	14.33	47
	EFC	-	12.54	17.83	87	-	8.44	9.84	55	-	6.22	6.76	38	-	4.17	2.81	11
	FAN	3,637	3.74	-	74	642	1.73	-	40	3,765	3.60	-	65	175	1.54	-	11
	LAA	7,280.08	0.97	-	1	1,421.11	0.98	-	1	6,254.97	0.98	-	1	430.44	0.95	-	1
	CDP	874.74	0.12	-	1	124.60	0.09	-	1	940.01	0.15	-	1	35.40	0.08	-	1
	CIT	2,320	0.31	-	35	255	0.18	-	19	2,273	0.35	-	30	74	0.16	-	6
Complexity	ABS	-	0.15	0.21	1	-	0.27	0.25	1	-	0.33	0.33	1	-	0.54	0.35	1
	WMC	12,903	17.68	27.37	346	2,628	9.77	11.27	68	10,537	14.38	21.92	262	498	6.30	6.30	37
	MCC	-	1.67	2.06	46	-	1.80	2.04	30	-	1.80	2.01	33	-	1.35	0.91	8
	WOC	581.22	0.60	-	1	178.43	0.48	-	1	658.82	0.63	-	1	74.10	0.65	-	1
	DBM	-	1.37	0.79	8	-	1.44	0.86	6	-	1.43	0.87	8	-	1.24	0.74	4
Inheritance	DIT	-	2.22	1.33	6	-	2.45	1.40	6	-	2.02	1.30	7	-	3.03	1.34	5
	NOH	493	0.68	3.77	69	147	0.55	1.54	11	337	0.46	1.82	28	59	0.75	2.05	10
	NRM	357	0.49	1.06	8	69	0.26	0.66	5	351	0.49	1.02	9	70	0.89	1.01	3

Table 1 Maintainability measures of Camel, Spring Integration, Mule, and Guaraná.

language (Frantz et al., 2011) . We refer to this language as Guaraná DSL. The implementation of the resulting models designed with Guaraná DSL into executable code is supported by a Software Development Kit (SDK), which includes a runtime system to support the execution of integration solutions as well. We refer to this kit as Guaraná SDK. In the following sections we introduce the domain-specific language and the software development kit. A commercial enterprise version of Guaraná is supported by i2Factory.

3.2 Measures

Table 1 summarises the results that we have collected. In this section, we provide an intuitive insight into them.

The architecture of the frameworks that we have analysed is organised into several packages: 54 in Camel, 32 in Spring Integration, and 124 in Mule. Although Mule has more than double as many packages as Camel, they have approximately the same total number of classes. Nevertheless, there are cases in which the maximum number of classes in a package reaches 96 in Camel, 58 in Spring Integration, and 51 in Mule. These values show that Camel has almost double as many classes in a package as Spring Integration or Mule. The same happens regarding the number of interfaces. Consequently, Camel has the highest standard deviation and mean values per package regarding both, classes and interfaces, which has an impact on the understandability of its packages. Spring Integration is the only framework that has a low value for the standard deviation regarding the number of interfaces. The architecture of Guaraná is organised into 18 packages, and the maximum number of classes in a package is no more than 11. Furthermore, Guaraná provides no more than 9 interfaces in these packages. The standard deviation computed for the number of classes and interfaces per package is very low, 3.09 and 0.76, respectively. These values indicate that maintenance in Guaraná is not expected to be difficult.

Other values that are impressive for these frameworks are regarding the total number of lines of code, which is very high, chiefly in the cases of Camel and Mule. These frameworks have $62,439$ and $67,090$ lines of code respectively, contrarily to $14,929$ in Spring Integration. The implementation of Guaraná has a total number of $2,878$ lines of code, which represents a big difference compared with the other frameworks. When analysing the methods in classes and interfaces, we found that Camel has $7,015$ methods compared to the $1,431$ and the $5,158$ found for Spring Integration and Mule, respectively. Most probably, the difference amongst Spring Integration and the other frameworks is because it has less than a half the number of classes and interfaces of Camel and Mule. The values that stand out are the maximum number of methods per class/interface computed in Camel and Mule, which are 192 and 129 respectively, contrarily to 39 in Spring Integration. Guaraná has 369 methods in total, with a maximum number of 24 methods per class/interface. If we look at the maximum number of parameters per method, it is also impressive how large it is, chiefly in Camel and Mule: 11 and 19 respectively. Spring Integration has a maximum of 9 parameters. These values indicate that some classes in Camel, Spring Integration, and Mule, are likely too application specific, with a limited possibility to be reused; furthermore, this makes some of their methods difficult to understand, chiefly in the case of Camel and Mule. Guaraná has no more than 4 parameters per method, which indicates that classes in Guaraná are expected to be more reusable and its methods not so difficult to understand.

Counting the number of lines of code inside methods, we found Camel has a total of $34,839$, Spring Integration has $8,264$, and Mule has $35,989$, which if compared to the total number of lines of code, represents 0.55%, 0.55%, and 0.53% of these values, respectively. It means there are many attributes declared in classes. The maximum value computed demonstrates that there are some methods with up to 141 lines of code in Camel, 110 in Spring Integration, and 180 in Mule. These values indicate that more effort might be necessary to maintain and understand the methods in these frameworks. Guaraná has a total number of $1,748$ lines of code inside methods, which, if compared to its total number of lines of code, represents 0.61% of this value. Furthermore, there is no method with more than 54 lines of code, being the average 4.72 lines of code per method. These values indicate that classes in Guaraná are expected to be easier to understand and maintain.

If we look at the number of static methods, Camel and Mule have a similar mean value per class, respectively 0.97 and 0.94. Contrarily, Spring Integration has a mean of 0.12 static methods per class. The difference between these frameworks is more evident when looking at the maximum number of static methods in a class. Whereas Camel and Mule have respectively 74 and 244, Spring Integration has 8. In Guaraná these values are incredible low; the maximum number of static methods is no more than 1, and the mean value is 0.01, which indicates that the code follows correctly the object-oriented paradigm. Considering the number of static attributes, there is also a big different amongst the analysed frameworks. Mule has an impressive number of 669 static attributes in total, whereas Camel and Spring Integration have 291 and 109, respectively. Such values indicate it must be difficult to reason about the state of these frameworks when testing has to be performed. Contrarily, Guaraná has only 30 static attributes in total, which indicates reasoning about its state is expected to be easier.

Regarding the number of attributes, the total values for Camel and Mule are still very high, $1,803$ and $1,417$, respectively. These values correspond to a mean of 2.47 and 1.93 attributes per class, reaching Camel the impressive number of 62 attributes in a single class.

Spring Integration has a total of 474 attributes, a mean of 1.76, and no more than 16 attributes in a class. In Guaraná the total number of attributes is 87, which corresponds to a mean of 1.10 attributes per class, this suggests that understanding the state of its classes is simpler that in Camel, Spring Integration, and Mule.

The mean and the maximum values for the lack of cohesion of methods is similar in every framework. Camel has 0.29 and 0.35, Spring Integration has 0.22 and 0.33, and Mule has 0.23 and 0.34. In Guaraná, the lack of cohesion of methods is very low, it presents a mean of only 0.14 in average. Regarding the coupling of classes, the values for the afferent and efferent coupling in every framework are very high. Camel has the highest value for the afferent coupling, followed by Mule and then Spring Integration, with a mean of 30.63, 12.69, and 22.90, respectively. The standard deviation is also very impressive, chiefly for Camel and Mule, which are 89.34 and 56.25, respectively. The maximum values are also very high, being 542 for Camel, 146 for Spring Integration, and 493 for Mule. These values suggest that much attention must be paid when performing maintenance in the classes of a package.

The mean for the efferent coupling varies from 12.54 in Camel and 8.44 in Spring Integration, to 6.22 in Mule. The maximum values are not so impressive as the afferent coupling, but they are still very high. In Camel, the maximum efferent coupling is 87; in Spring Integration, it is 55; in Mule, it is 38. These figures suggest that the classes inside a package have a large number of dependencies on external classes and maintenance has to be done carefully; as a conclusion, the impact on maintenance should not be neglected at all. Regarding the coupling of classes, the values for the afferent and efferent coupling in Guaraná are not very high. The afferent coupling has values 6.94, 14.33, and 47 as mean, standard deviation, and maximum, respectively. The efferent coupling has values 4.17, 2.81, and 11 as mean, standard deviation, and maximum, respectively. The average afferent and efferent couplings in Guaraná are 15.13 and 4.90 less than in the other frameworks, respectively. These values suggest that the classes in Guaraná do not have a high number of dependencies and maintenance is expected to be easy.

Considering the number of called classes, once more Camel and Mule have very high values, compared to Spring Integration, respectively 3, 637, 3, 765, and 642. If we look at the maximum number of calls a class receives, Camel has 74, Spring Integration 40, and Mule 65. In Guaraná the total number of called classes is 175 and the maximum number is no more than 11. These values indicate that method calls in Guaraná are not complex. The locality of attribute accesses is similar in every framework. If we consider the mean value, Camel, Spring Integration, and Mule have 0.97, 0.98, and 0.98, respectively. The mean in Guaraná is lower, 0.95. Regarding the coupling dispersion, the mean value indicates that Mule has the highest dispersion with 0.15, followed by Camel and Spring Integration, respectively with 0.12 and 0.09. Mule has also a very high value in total, 940.01, compared to Camel and Spring Integration with 874.74 and 124.60, respectively. These values indicate that Mule has an improper distribution of functionality amongst its methods. The mean value in Guaraná is 0.08, which ranks it close to Spring Integration. If we look at the maximum values for the coupling intensity of these frameworks, these values demonstrate an excessive coupling amongst the methods in these frameworks, since the values in Camel, Spring Integration, and Mule are 35, 19, and 30, respectively. Contrarily, in Guaraná the maximum value is 6, which indicates a low coupling amongst its methods.

The values for the degree of abstractness indicates that Camel is the less abstract framework. The mean value for Camel is 0.15, followed by 0.27 for Spring Integration, and 0.33 for Mule. The results indicate that these frameworks are not as easy to customise, chiefly Camel because its mean value is very low. The degree of abstractness in Guaraná is very high. Its mean value is 0.54, which ranks it 0.29 in average more abstract than the other frameworks. These values suggest that Guaraná is more abstract and then it is expected to be easier to adapt to specific domains.

The weighted method complexity computed also demonstrates a high cyclomatic complexity within classes, chiefly for Camel and Mule. In these frameworks, the total weighted method complexity was 12, 903 and 10, 537, respectively. For Spring Integration, the cyclomatic complexity is 2, 628, which is not so high when compared to Camel and Mule. Nevertheless, not only the total cyclomatic complexity is high, but also the mean, the standard deviation, and the maximum. Camel, Spring Integration, and Mule have maximum values of 346, 68, and 262, respectively. In Guaraná, the total value is 498, the mean and the standard deviation is 6.30, and the maximum is 37. These values indicate a low cyclomatic complexity within the classes of Guaraná.

The values computed for the McCabe cyclomatic complexity indicate that there are cases in which it is extremely high. This is indicated by the maximum values, which reach 46, 30, and 33 in Camel, Spring Integration, and Mule, respectively. Consequently, they are also very complex frameworks, which may have a serious impact on their maintenance. The values computed for the McCabe cyclomatic complexity indicate that the maximum value in Guaraná is 8, which amounts to 28.33 less complexity than other frameworks. These values indicate the architecture in Guaraná is well designed and maintenance is expected to be easier.

The mean value for the weight of classes indicates that classes in Spring Integration are complex. The mean value for Spring Integration is 0.48, followed by 0.60 for Camel, and 0.63 for Mule. In Guaraná, the mean value is 0.65, which indicates that classes in Guaraná are not too complex. The depth of nested blocks in a method is similar in every framework. If we consider the mean and maximum values, Camel has 1.37 and 8, Spring Integration has 1.44 and 6, and Mule has 1.43 and 8, respectively. In Guaraná, the mean and maximum values for the depth of nested blocks is 4 and 1.24, respectively. These values indicate that debugging a piece of code in Guaraná is not expected to be as difficult as in the other frameworks.

The depth of inheritance tree in Mule has a maximum value of 7, which makes it more complicated to maintain a class in this framework. Camel and Spring Integration have equal values, 6. In Guaraná, the maximum value is not greater than 5. The maximum number of immediate children classes of a class also varies very much: 69 in Camel, 11 in Spring Integration, and 28 in Mule. When the mean and the standard deviation values per class, Camel has the highest values, which indicates that the abstraction defined by parent classes tend to be poorly designed.

The maximum number of immediate children classes of a class in Guaraná is not greater than 10, with a mean of 0.75 per package. These values indicate that the abstraction defined by the parent class is well designed in Guaraná. Regarding the number of overriden methods, Spring Integration has the lowest mean value amongst the analysed frameworks, and Camel and Mule have the same value, respectively with 0.26, 0.49, and 0.49. In Guaraná, the mean value is 0.89, which indicates that the classes in this framework are more adaptable than in Camel, Spring Integration, and Mule.

Tool	Total	Mean
Guaraná	1.56	1.24
Spring Integration	2.56	2.08
Camel	2.64	3.16
Mule	3.24	3.52

Table 2 Empirical Rankings.

Test	Total	Mean
Statistic	9.84	44.18
P-value	1.61E-5	3.33E-16

Table 3 Results of Iman-Davenport's test.

Comparison	Statistic	ap-value	Tool	Rank
Mule vs. Guaraná	4.60	2.52E-5	Guaraná	1
Camel vs. Guaraná	2.95	9.29E-3	Spring Integration, Camel, Mule	2
Spring Integration vs. Guaraná	2.73	0.01	-	-
Spring Integration vs. Mule	1.86	0.18	-	-
Camel vs. Mule	1.64	0.18	-	-
Camel vs. Spring Integration	0.21	0.82	-	-

a) Total values

Comparison	Statistic	ap-value	Tool	Rank
Mule vs. Guaraná	6.24	2.56E-9	Guaraná	1
Camel vs. Guaraná	5.26	4.36E-7	Spring Integration	2
Spring Integration vs. Mule	3.94	2.40E-4	Camel, Mule	3
Camel vs. Spring Integration	2.96	3.10E-3	-	-
Spring Integration vs. Guaraná	2.30	4.28E-2	-	-
Camel vs. Mule	0.98	3.24E-1	-	-

b) Mean values

Table 4 Results of Bergmann-Hommel's test.

3.3 Analysis

Table 2 shows the empirical rankings that we got; note that Guaraná ranks the first regarding both the total and the mean values. Then, we used Iman-Davenport's test to check if there are statistically significant differences in these ranks at the standard significance level ($\alpha = 0.05$). Table 3 shows the results; note that the p-value is largely smaller than the standard significance level, which is a strong indication that the empirical ranks are different from a statistical point of view.

As a conclusion, it makes sense to perform Bergmann-Hommel's test to rank every pair of proposals. Table 4 shows the results. Regarding the total measures, note that the comparisons of Guaraná with the other techniques results in adjusted p-values (ap-values) that are always significantly smaller than the significance level, which is a strong indication that Guaraná's measures are better than the others; note, too, that the adjusted p-values that correspond to the remaining comparisons are not smaller than the significance level, which indicates that there are not any significant differences amongst the measures of the other frameworks. Regarding the mean measures, the results are similar; the only difference is that Guaraná is significantly better than Spring Integration, which, in turn, is significantly better than both Camel and Mule at the standard significance level.

As a conclusion, and based on these four statistical tests, there is enough statistical evidence in the measures that we have collected to indicate that Guaraná outperforms the other integration frameworks regarding maintainability of the core implementation.

4 Conclusions

Companies that provide Enterprise Application Integration solutions are interested in Enterprise Application Integration frameworks that can be easily adapted to focus on specific contexts. We have assembled a collection of 25 measures in the literature that provide an overall overview of how easy it is to adapt a system (Lanza and Marinescu, 2006; Lajios, 2009; Herraiz et al., 2009; Risi et al., 2013; Li and Henry, 1993; Sheldon et al., 2002; Bocco et al., 2005; Mouchawrab et al., 2005; Briand et al., 1998; Chidamber and Kemerer, 1994; Henderson-Sellers, 1996; Martin, 2002; McCabe, 1976) and we have also proposed a statistically sound methodology to analyse the results.

We have also illustrated our methodology in industry by comparing Camel, Spring Integration, Mule, and Guaraná, which range amongst the most recent and important open-source integration frameworks based on integration patterns (Hohpe and Woolf, 2003). The sample application of our methodology has considered only the core implementation of the analysed integration frameworks, which is a general-purpose core. The core of Camel, Spring Integration, Mule, and Guaraná provide support for the same functionalities and all of them address multiple domains. Note that, in our research, we do not take into account the code required to implement the adapters in these four integration frameworks, because it is peripheral and, more often than not, comes from other open-source projects that are maintained separately, otherwise the comparison would be totally unfair. Thus the main limitation of our proposal is that software engineers can use it to compare only those parts of integration frameworks that are equivalent in terms of functionality and that have a similar architectural style. Integration frameworks differ one from another, mainly regarding the number of adapters and other features that help software engineers in the design, the implementation and in monitoring the solutions. This is the reason why most often it is not possible to take into account all the features and thus the whole integration framework in a comparison. Another limitation may be the tool support to compute the maintainability measures in an automated fashion, since the software tools we found in the literature take as input only source code written with the Java language, which would limit the the scope of integration frameworks that could be analysed. As future work, we plan to extend the number of measures to improve the accuracy of our methodology, chiefly measures that deal with size and complexity. More research should be conducted in the direction to endow our methodology with the possibility to compare integration frameworks considering their difference in terms of feature, size, and architecture.

Acknowledgments

The work in this article was supported by the Evangelischer Entwicklungsdienst e.V. (EED), the Spanish and the Andalusian R&D&I programmes and the European Commission (FEDER) (grants: TIN2007-64119, P07-TIC-2602, P08-TIC-4100, TIN2008-04718-E, TIN2010-21744, TIN2010-09809-E, TIN2010-10811-E, TIN2010-09988-E, and TIN2013-40848-R).

References

Alkadi, G. and Alkadi, I. (2003). Application of a revised DIT metric to redesign an OO design. *Journal of Object Technology*, 2(1):127–134.

Antonellis, P., Antoniou, D., Kanellopoulos, Y., Makris, C., Theodoridis, E., Tjortjis, C., and Tsirakis, N. (2007). A data mining methodology for evaluating maintainability according to ISO/IEC-9126 software engineering-product quality standard. In *Proc. special sessions in IEEE 11th European Conf. on Software Maintenance and Reengineering*, pages 81–89.

Balmas, F., Bergel, A., Denier, S., Ducasse, S., Laval, J., Mordal-Manet, K., Abdeen, H., and Bellingard, F. (2009). Software metric for Java and C++ practices. Technical report, French Institute for Research in Computer Science and Automation.

Bergin, S. and Keating, J. (2003). A case study on the adaptive maintenance of an Internet application. *Journal of Software Maintenance*, 15(4):254–264.

Bocco, M. G., Piattini, M., and Calero, C. (2005). A survey of metrics for UML class diagrams. *Journal of Object Technology*, 4(9):59–92.

Briand, L. C., Daly, J. W., and Wüst, J. (1998). A unified framework for cohesion measurement in object-oriented systems. *Empirical Software Engineering*, 3(1):65–117.

Briand, L. C., Daly, J. W., and Wüst, J. (1999). A unified framework for coupling measurement in object-oriented systems. *IEEE Trans. Software Eng.*, 25(1):91–121.

Burger, S. and Hummel, O. (2012). Applying maintainability oriented software metrics to cabin software of a commercial airliner. In *CSMR*, pages 457–460.

Chae, H. S., Kim, T. Y., Jung, W.-S., and Lee, J.-S. (2007). Using metrics for estimating maintainability of web applications: An empirical study. In *ACIS-ICIS*, pages 1053–1059.

Chalmeta, R. and Pazos, V. (2015). A step-by-step methodology for enterprise interoperability projects. *Enterp. Inf. Syst.*, 9(4):436–464.

Chen, J.-C. and Huang, S.-J. (2009). An empirical analysis of the impact of software development problem factors on software maintainability. *Journal of Systems and Software*, 82(6):981–992.

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

Coleman, D. M., Ash, D., Lowther, B., and Oman, P. W. (1994). Using metrics to evaluate software systems maintainability. *IEEE Computer*, 27(8):44–49.

Demšar, J. (2006). Statistical comparisons of classifiers over multiple data sets. *Journal of Machine Learning Research*, 7:1–30.

Dong, X. and Godfrey, M. W. (2009). Understanding source package organization using a hybrid model. In *International Conference on Software Maintenance*.

Dossot, D. and D'Emic, J. (2009). *Mule in Action*. Manning.

Epping, A. and Lott, C. M. (1994). Does software design complexity affect maintenance effort? In *NASA/Goddard 19th Annual Software Engineering Workshop*, pages 297–313.

Fisher, M., Partner, J., Bogoevici, M., and Fuld, I. (2010). *Spring Integration in Action*. Manning.

Fowler, M. (2010). *Domain-Specific Languages*. Addison-Wesley.

Frantz, R. Z. and Corchuelo, R. (2012). A software development kit to implement integration solutions. In *27th Symposium On Applied Computing*, pages 1647–1652.

Frantz, R. Z., Reina-Quintero, A. M., and Corchuelo, R. (2011). A Domain-Specific language to design enterprise application integration solutions. *International Journal of Cooperative Information Systems*, 20(2):143–176.

García, S., Fernández, A., Luengo, J., and Herrera, F. (2010). Advanced nonparametric tests for multiple comparisons in the design of experiments in computational intelligence and data mining: Experimental analysis of power. *Inf. Sci.*, 180(10):2044–2064.

Genero, M., Olivas, J. A., Piattini, M., and Romero, F. P. (2001). Using metrics to predict OO information systems maintainability. In *CAiSE*, pages 388–401.

He, W. and Xu, L. D. (2014). Integration of distributed enterprise applications: A survey. *Industrial Informatics, IEEE Transactions on*, 10(1):35–42.

Heitlager, I., Kuipers, T., and Visser, J. (2007). A practical model for measuring maintainability. In *6th International Conference on the Quality of Information and Communications Technology*, pages 30–39.

Henderson-Sellers, B. (1996). *Object-Oriented Metrics, Measures of Complexity*. Prentice Hall.

Herraiz, I., Izquierdo-Cortazar, D., and Rivas-Hernández, F. (2009). Flossmetrics: Free/libre/open source software metrics. In *CSMR*, pages 281–284.

HIPAA (2011). Health insurance portability and accountability act.

Hitt, L. M., WU, D. J., and Zhou, X. (2002). Investment in enterprise resource planning: Business impact and productivity measures. *J. Manage. Inf. Syst.*, 19(1):71–98.

HL7 (2011). Health level seven international.

Hohpe, G. and Woolf, B. (2003). *Enterprise Integration Patterns - Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley.

Ibsen, C. and Anstey, J. (2010). *Camel in Action*. Manning.

Jorgensen, M. (1995). An empirical study of software maintenance tasks. *Journal of Software Maintenance*, 7(1):27–48.

Kumar, D. S., Prasad, R. S., and and, R. (2015). Construction and testing of polynomials predicting software maintainability. *International Journal of Advanced Research in Computer Science Engineering and Information Technology*, 5(3):384–396.

Lajios, G. (2009). Software metrics suites for project landscapes. In *CSMR*, pages 317–318.

Lanza, M. and Marinescu, R. (2006). *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer.

Li, W. and Henry, S. M. (1993). Object-oriented metrics that predict maintainability. *Journal of Systems and Software*, 23(2):111–122.

Lorenz, M. and Kidd, J. (1994). *Object Oriented Software Metrics*. Prentice Hall.

Marinescu, R. (2002). *Measurement and Quality in Object-Oriented Design*. PhD thesis, Department of Computer Science, Politehnica University of Timisoara.

Martin, R. C. (2002). *Agile Software Development, Principles, Patterns, and Practices*. Prentice Hall.

McCabe, T. J. (1976). A complexity measure. *IEEE Trans. Software Eng.*, 2(4):308–320.

Messerschmitt, D. and Szyperski, C. (2003). *Software EcoSystemm: Understanding an Indispensable Technology and Industry*. MIT Press.

Metrics (2015). Metrics 1.3.6 Tool Home Page.

Mordal-Manet, K., Anquetil, N., Laval, J., Serebrenik, A., Vasilescu, B., and Ducasse, S. (2013). Software quality metrics aggregation in industry. *Journal of Software: Evolution and Process*, 25(10):1117–1135.

Mouchawrab, S., Briand, L. C., and Labiche, Y. (2005). A measurement framework for object-oriented software testability. *Information & Software Technology*, 47(15):979–997.

Nair, T. R. G., Aravindh, B. S., and Selvarani, R. (2010). Design property metrics to maintainability estimation: a virtual method using functional relationship mapping. *ACM SIGSOFT Software Engineering Notes*, 35(6):1–6.

Offutt, J., Abdurazik, A., and Schach, S. R. (2008). Quantitatively measuring object-oriented couplings. *Software Quality Journal*.

Oman, P. W. and Hagemeister, J. R. (1994). Construction and testing of polynomials predicting software maintainability. *Journal of Systems and Software*, 24(3):251–266.

Risi, M., Scanniello, G., and Tortora, G. (2013). Metric attitude. In *CSMR*, pages 405–408.

RosettaNet (2011). RosettaNet.

Schneidewind, N. F. (1987). The state of software maintenance. *IEEE Trans. Software Eng.*, 13(3):303–310.

Sheldon, F. T., Jerath, K., and Chung, H. (2002). Metrics for maintainability of class inheritance hierarchies. *Journal of Software Maintenance*, 14(3):147–160.

Sheskin, D. J. (2012). *Handbook of Parametric and Nonparametric Statistical Procedures*. Chapman and Hall/CRC, 5th edition.

Swift (2011). Society for worldwide interbank financial telecommunication.

Tempero, E. D., Noble, J., and Melton, H. (2008). How do Java programs use inheritance? An empirical study of inheritance in Java software. In *ECOOP*, pages 667–691.

Thwin, M. M. T. and Quah, T.-S. (2003). Application of neural networks for estimating software maintainability using object-oriented metrics. In *SEKE*, pages 69–73.

Yu, L. (2008). Common coupling as a measure of reuse effort in kernel-based software with case studies on the creation of MkLinux and Darwin. *Journal of the Brazilian Computer Society*, 14:45–55.

1. http://metrics.sourceforge.net

2. http://loose.upt.ro/iplasma