A Measure to Assess the Behavior of Method Stereotypes in Object-Oriented Software

Peter Andras, Anjan Pakhira
School of Computing Science
Newcastle University
Newcastle upon Tyne, UK
peter.andras@ncl.ac.uk, anjan.pakhira@ncl.ac.uk

Laura Moreno, Andrian Marcus
Department of Computer Science
Wayne State University
Detroit, MI, USA
lmoreno@wayne.edu, amarcus@wayne.edu

***Abstract*—The implementation of software systems should ideally follow the design intentions of the system. However, this is not always the case – the design and implementation of software systems may diverge during software evolution. In this paper we propose a measure based on run time information to assess the consistency between the design and the implementation of OO methods. The measure is based on the analysis of the run-time behavior of methods and considers the frequency of fan-in and fan-out method calls. We analyze this measure with respect to the design intent of methods, reflected by their stereotype. We apply the proposed approach to data from three open source software systems and analyze the behavior of method stereotypes across the systems and within each system. The analysis shows that most methods behave as expected based on their stereotypes and it also detects cases that may need re-engineering attention.**

***Index Terms*— Software design, dynamic analysis, dynamic metric, method stereotypes.**

I. Introduction

Designing, implementing, and maintaining large software systems are complex software engineering activities. One of the sources of the problems in these activities is the mismatch between the design intent and the actual behavior of software [1, 2]. For instance, the intentions behind the design of a method imply expectations about its run-time behavior, which, in turn, should match its designed behavior. Measuring this matching is problematic in itself.

Method stereotypes [3] capture the intent of methods in Object Oriented (OO) software systems based on static analysis. Such stereotypes (e.g., *get*, *factory*, *constructor*) are characterized by their access to data (e.g., reading or writing data) and their main designed behavioral features (e.g., *creational*, *structural,* and *collaborational* methods). In this sense, the method stereotype definitions imply certain run-time behavior of the methods. However, the expected run-time behavior of these methods is not enforced by any programming language construct. Hence mismatches between the design intent and the run-time behavior of the software occur.

We address this issue by describing a measure-based approach to compare the usage of method stereotypes across software systems. Our conjecture is that if the run-time behavior of methods is captured by some measure, then the distribution of methods of the same stereotype according to this measure is similar across different software systems and the distributions corresponding to different method stereotypes are different within a given software system. Such a measure could be used then to assess to which extent the design of a system (or at least part of it) matches its behavior.

This paper has two key contributions:

(1) the definition of the dynamic *fI-fO* distribution measure for method stereotypes; the measure is based on the analysis of the run-time behavior of methods considering the frequency of fan-in and fan-out method calls (i.e., other methods calling the measured method and other methods called by the method);

(2) the application of this measure to show that indeed, in the case of a small group of well-designed and well-implemented software systems, the dynamic *fI-fO* distribution measure captures the similarity of the implementations of the same method stereotypes across the software systems and also the difference between different method stereotypes within a single software system.

II. Related Work

Considerable research has been dedicated to the extraction of design elements from the source code. Special attention has been given to the detection of design patterns either by static [4, 5], dynamic [6], or hybrid analysis [7]. Few researchers have explored the identification of the design intent at lower levels of abstraction. In this regard, common programming constructs of Java classes are gathered in [8] to form a set of *micro patterns*, which are extracted from software binaries. In a similar way, a catalog of *nano patterns* is described in [9], which correspond to the characterization of Java methods based on their structural properties. These approaches are extended by *class* [10] and *method stereotypes* [3], which are, respectively, categories for describing the intent of classes in a system's design and the responsibility of methods within a class, based on implementation rules.

Dynamic analysis has been widely used in program comprehension research. A recent survey on this [11] revealed that almost 70% of the work on dynamic analysis has focused on OO programs, and basic visualization methods (e.g., graphs or diagrams). Dynamic method call graphs, in particular, have been used for program slice analysis [12] and feature location [13].

Different aspects of software systems have been assessed through dynamic metrics [14-20]. These metrics usually focus on object or class coupling, cohesion, and graph complexity [16, 17] during software execution. Some of them have been used to assess the quality of the implemented software [14, 15, 19, 21, 22]. However, the validity of some of these metrics (e.g. lack of cohesion metric) has been questioned [23].

The mismatch between the design and implementation of the software has been the subject of several studies on the quality of the software. Design bad smells [24] are one manifestation of such mismatch. Lutz [2] investigated the impact of mismatch between design intentions and implementation in the context of software safety. Feather et al. [1] analyzed the match between requirements and the run-time behavior of the software. Garlan et al. [25, 26] considered the impact of architectural mismatch on software quality in the context of re-use of software components.

III. Static Method Stereotypes

Code stereotypes are low-level categories that reveal the intention of source code artifacts based on implementation patterns, i.e., by static analysis. In the case of methods, stereotypes represent their general responsibility within a class [3]. For example, the stereotype *get* describes a method that returns a class's field, without modifying any value or invoking other methods.

In this work, we use the method stereotype taxonomy for Java code [27], adapted from previous work [3]. This taxonomy defines 17 stereotypes classified as follows: *structural*, when the main purpose of the method is to retrieve (*accessors*) or to modify (*mutators*) the class' fields; *creational* if the method is responsible for creating or destroying objects; *collaborational* when the method communicates or controls objects in the system; and *degenerate*, in any other case. A short description of each stereotype is provided in Table 1 based on [3, 27]. It is important to mention that methods have a primary stereotype from any category and an optional, secondary stereotype in the *collaborational* category.

IV. The Dynamic *fI-fO* Distribution Metric for Method Stereotypes

We define the dynamic *fI-fO* classification for a given method by considering the number and the frequency of methods that call the given method (fan-in) and those that are called by the given method (fan-out) during run time. We use a simple classification of the number of fan-in and fan-out methods, by considering as separate categories *0*, *1*, *few* (2 – 4), and *many* (5 or more) methods – this is a natural classification of the fan-in and fan-out numbers given their actual distributions. This way, for a given method we get a fan-in-fan-out (*fI-fO*) category that combines two of the above defined counting categories. For example, the *1-few* category denotes a method that is called by a single method and calls 2-4 other methods. Since each method has to be called at least once to be executed, on the fan-in side the available options will be only *1*, *few*. and *many*.

We also consider the run-time frequencies of the method calls in the cases when there are *few* or *many* methods on the fan-in or fan-out side. We define a call distribution as *balanced* if it is close to the uniform distribution, and *unbalanced* when it is considerably different from the uniform distribution. To measure the difference between the distributions we use the Kullback-Leibler (KL) divergence. According to this measure if there are n methods that call (or are called by) the given method and the frequency of calls from these methods are f_i with $i = 1,...,n$, then the KL divergence of the calls distribution relative to the uniform distribution is calculated as:

$$d = \ln(n) + \sum_{i=1}^{n} f_i \cdot \ln(f_i)$$

If the method call distribution is uniform, then $d = 0$. We consider a method call distribution *balanced* if $d < 0.3$ and *unbalanced* otherwise. Therefore, for each method the dynamic *fI-fO* classification assigns one of the 30 categories listed in Table 2.

Considering the method stereotype taxonomy summarized in Table 1 and given a software system, we extract the set of methods that belong to each stereotype. Then, we determine the dynamic *fI-fO* category for each method in the system. With this information we are able to find the distribution of methods over the 30 dynamic *fI-fO* categories for each method stereotype. Such a distribution characterizes the method stereotype within the considered software system.

TABLE 1. METHOD STEREOTYPE TAXONOMY

Category		Stereotype	General description
Structural	Accessor	Get	Returns a local field directly
		Predicate	Returns a Boolean value that is not a local field
		Property	Returns information about local fields
		Void-accessor	Returns information about local fields through the parameters
	Mutator	Set	Changes only one local field
		Command	Changes more than one local fields
		Non-void command	Command whose return type is not void or Boolean
Creational		Constructor	Invoked when creating an object
		Destructor	Performs any necessary cleanups before the object is destroyed
		Copy-constructor	Creates a new object as a copy of the existing one
		Factory	Instantiates an object and returns it
Collaborational		Collaborator	Connects one object with other type of objects
		Controller	Provides control logic by invoking only external methods
		Local-controller	Provides control logic by invoking only local methods
Degenerate		Abstract	Has no body
		Empty	Has no statements
		Incidental	Any other case

TABLE 2. DYNAMIC FI-FO CATEGORIES FOR METHODS

fI \ fO	0	1	F	M
1	1-0	1-1	1-FB 1-FU	1-MB 1-MU
F	FB-0 FU-0	FB-1 FU-1	FB-FB FB-FU FU-FB FU-FU	FB-MB FB-MU FU-MB FU-MU
M	MB-0 MU-0	MB-1 MU-1	MB-FB MB-FU MU-FB MU-FU	MB-MB MB-MU MU-MB MU-MU

Notation (*fI-fO*)

0 = no method call
1 = a single method call
FB = a few (2, 3,or 4), balanced methods calls
FU = a few (2, 3,or 4), unbalanced methods calls
MB = many (5 or more), balanced methods calls
MU = many (5 or more), balanced methods calls

For example, one may consider that a method categorized as *get* stereotype would typically belong to one of the *1-0*, *FB-0*, *MB-0* dynamic *fI-fO* categories, i.e., there are one or more uniformly distributed calls from other methods and no calls going out to other methods. However, in practice, most likely the distribution over *fI-fO* categories of the methods belonging to the *get* stereotype will not be restricted to the dynamic *fI-fO* categories *1-0*, *FB-0*, *MB-0*, but will cover other dynamic *fI-fO* categories. Then this is the characteristic distribution of *get* method stereotype for a given software system.

Having a distribution over the dynamic *fI-fO* categories for each method stereotype allows us to compare these distributions in the context of a software system. It also allows us to compare the distributions associated with the same method stereotype in the context of different software systems. The comparison of the distributions can be done using the KL divergence for the considered distributions (note that the KL divergence is not symmetric, i.e., the divergence measure of distribution D relative to D' is usually not the same as the divergence of D' relative to D). Let D and D' be the distributions over the dynamic *fI-fO* categories associated with two method stereotypes:

$$D = \{f_{lb}, lb \in \text{fI - fO class labels}\}$$
$$D' = \{f'_{lb}, lb \in \text{fI - fO class labels}\}$$

The KL divergence of D relative to D' is

$$d = \sum_{lb \in \text{fI-fO category labels}} f_{lb} \cdot \left(\ln(f_{lb}) - \ln(f'_{lb})\right)$$

which defines the dynamic *fI-fO* distribution measure for method stereotypes comparison within a software system and across software systems.

In principle, we may expect that if the implementation of a software system follows some design intent, then the distribution over *fI-fO* categories that corresponds to a given method stereotype should be close to an ideal (or intended) distribution over the *fI-fO* categories. The dynamic *fI-fO* distribution measure based on the KL divergence of distributions can be used as a measure to quantify the extent to which the actual behavior of the software follows the design intent expressed by the choice of method stereotypes. In this sense, large divergence values between distributions indicate high mismatch between the design intent of methods and their run-time behavior, whereas small divergence values are an indicator of low mismatch.

Similarly, it may be assumed that different method stereotypes have different associated distributions over the *fI-fO* categories within a single software system. This reflects the expectation that methods with different method stereotypes should behave differently during run time (e.g., *set* methods should have a different run time behavior from *controller* methods in terms of incoming and outgoing method calls). Having the measured *fI-fO* category distributions associated with method stereotypes for a software system, we can compare these distributions and assess their difference using the dynamic *fI-fO* distribution measure. Assuming that in an ideal (or intended) implementation of the software we might expect a certain level of difference between the distributions over the *fI-fO* categories, the comparison of the actual distributions can provide a measure of the extent to which these differences are confirmed. If the differences are less than expected, they might indicate that the implementation of methods belonging to distinct method stereotypes is not sufficiently different across the software. This might be a marker of poor design (or implementation) in the software system. The opposite, of course, might be considered as an indicator of good design (or implementation).

Thus, the proposed dynamic *fI-fO* distribution measure for method stereotypes can be used to assess software systems, regarding to the extent in which the implemented behavior of methods matches the intended behavior of the software specified by the design.

V. CASE STUDY

We conducted a case study in order to verify our assumptions about the expected behavior of method stereotypes.

A. Definition and design

We formulated two research questions:

RQ1: Are the behavior of methods of different stereotypes similar or different?

RQ2: Are the behavior of methods of the same stereotype consistent across different software?

The *context* of our study is represented by the source code and execution traces of three open-source Java software systems: ArgoUML (version 0.22, 924 KLOC), a UML modeling tool; muCommander (version 0.8.5, 85 KLOC), a file manager; and JabRef (version 2.6, 148 KLOC), a bibliography reference manager. We chose these systems for three reasons: (i) they are used in existing benchmarks in software maintenance research [28]; (ii) their domains and sizes are different; and (iii) they have a set of publicly available

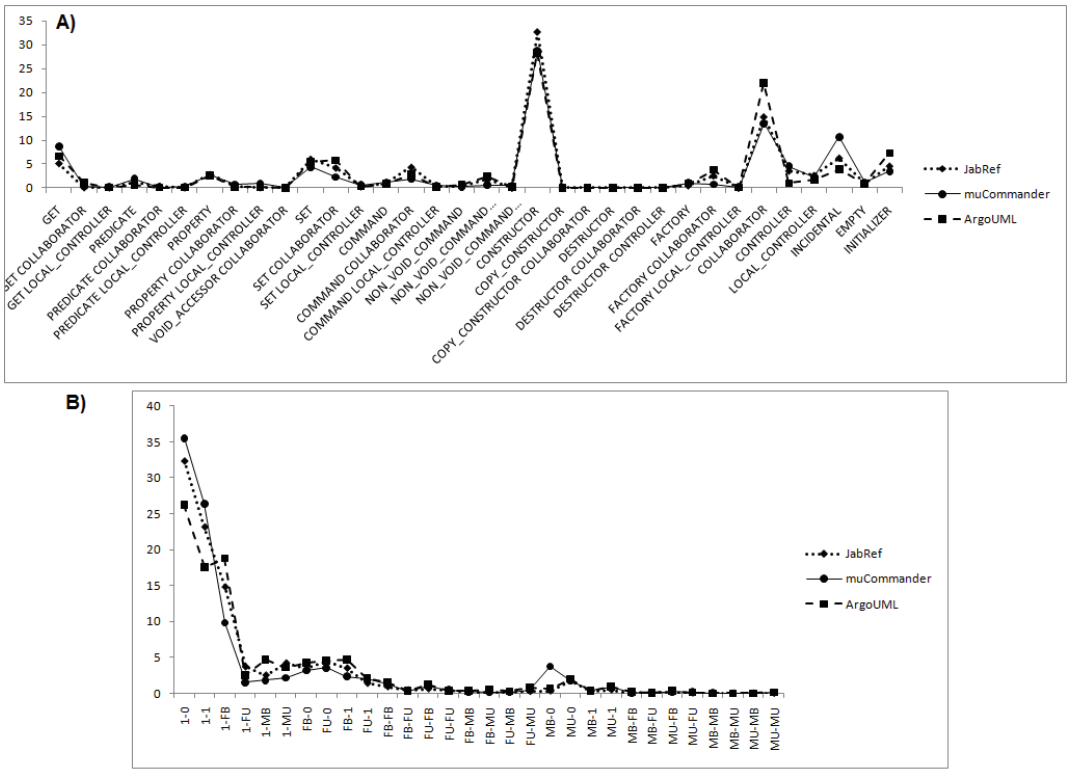

Fig. 1. Distribution of (a) method stereotypes and (b) dynamic *fI-fO* category in three Java systems. In both cases the vertical axis shows the percentage of methods belonging to the method stereotypes and dynamic *fI-fO* categories, respectively, shown on the horizontal axis.

execution traces[1]. Each set of traces corresponds to the execution of several scenarios for each system. The traces are XML files generated by the Eclipse Test & Performance Tools Platform (TPTP), and they capture every method executed in a given scenario since the program is started until it is ended.

B. Procedure

In order to extract the method stereotypes we used JStereoCode [27], an Eclipse plug-in that identifies code stereotypes from Java systems. The tool produced a classification consisting of 34 method stereotypes (including combined stereotypes), which we considered in the rest of the study.

For each system, we used the corresponding set of traces to classify the executed methods as one of the 30 dynamic *fI-fO* categories listed in Table 2.

We merged the stereotype and the *fI-fO* information to calculate the distribution of methods over the dynamic *fI-fO* categories for each method stereotype. The calculated distributions were averaged over all traces per software system to get robust estimates of the distributions. The resulting distributions over dynamic *fI-fO* categories were used for further analysis.

To assess whether the difference between two distributions, according to the dynamic *fI-fO* distribution measure, was large or small we needed to generate a large set of distributions of the same kind with random parameters. With this large set of randomly chosen distributions of the right kind, we could calculate the expected value of their difference according to the dynamic *fI-fO* distribution measure and also the standard deviation of these measured differences. Our expected value (E) and standard deviation (σ) calculations are based on a random selection of distributions and we assume that the values of the measured differences are distributed normally. Accordingly, a measured difference value calculated for two stereotype distributions over the dynamic *fI-fO* categories is *small* if the value is less than $L_1 = E - 2.33\sigma$ (i.e., the lowest 1% range of the distribution of the measured difference values), which means that it is much smaller than what is expected for two randomly chosen distributions. The difference value is *large* if it is greater than $L_2 = E - 1.96\sigma$ (i.e., above the lowest 2.5% range of the measured difference value distribution, and within the 95% range of the value distribution) or, in other words, if the value is in the expected wide range for two randomly chosen distributions. If the measured difference value calculated for two stereotype distributions is in the intermediary range $[E - 2.33\sigma, E - 1.96\sigma]$, the value is considered *moderately large*.

C. Results

After analyzing the dynamic *fI-fO* category distributions corresponding to method stereotypes, we found that the frequency values of these distributions follow log-normal distributions with parameters (m, s) with values in the ranges of $[-2,0]$ for m and $[2.2,2.8]$ for s. That is, the distributions

[1] http://www.cs.wm.edu/semeru/data/benchmarks/

TABLE 3. KL DIVERGENCES FOR DISTRIBUTIONS OVER METHOD STEREOTYPES

Software vs software	KL divergence
ArgoUML vs muCommander	0.11772
muCommandervs ArgoUML	0.24544
JabRef vs muCommander	0.01071
muCommandervs JabRef	0.10955
ArgoUML vs JabRef	0.06368
JabRef vs ArgoUML	0.05051

TABLE 4. KL DIVERGENCES FOR DISTRIBUTIONS OVER DYNAMIC FI-FO CATEGORIES

Software vs software	KL divergence
ArgoUML vs muCommander	0.13181
muCommandervs ArgoUML	0.12799
JabRef vs muCommander	0.09463
muCommandervs JabRef	0.07578
ArgoUML vs JabRef	0.05290
JabRef vs ArgoUML	0.06574

of the frequency values are such that their logarithms are normal distributions with mean *m* and standard deviation *s*.

In accordance with the previously described procedure for the estimation of the values of E and σ, we generated 100 randomly chosen distributions using frequency values provided by log-normal distributions with parameters in the above given ranges. We calculated the dynamic *fI-fO* distribution measure for pairs of these distributions and found the values of E and σ to be $E = 4.4838$ and $\sigma = 1.1817$. Consequently, a measured difference for two stereotype distributions is considered *small* if it is below $L_1 = 0.2628$, *large* if it is above $L_2 = 0.9328$, and *moderately large* if it is in the interval of $[L_1, L_2] = [0.2625, 0.9328]$.

We used the KL divergence to assess the overall similarity of the three software systems that we considered by comparing the distributions of methods over method stereotypes and also over the dynamic *fI-fO* categories. Fig. 1 shows the two sets of distributions for the three considered software. Table 3 and Table 4 show the calculated pair-wise KL divergence values. All the values are below L_1, indicating that the software systems are, in general, similar to each other in terms of the distribution of their methods over method stereotypes and dynamic *fI-fO* categories.

Next, we compared the method stereotypes using the dynamic *fI-fO* distribution measure in the context of each software system considered in the study. The results for method stereotypes to which a larger number of methods belong to (at least 1% of all methods) in ArgoUML are shown in Table 5. Fig. 2a shows the distributions over dynamic *fI-fO* categories for three method stereotypes: *collaborator*, *constructor*, and *initializer*. The calculated measure values shows that 32% of these are above L_2, 42% are in the range of moderately large values $[L_1, L_2]$, and 16% are smaller than L_1. This confirms that the implementations of different method stereotypes within one software system are different in the large majority of the cases, as it is expected in a software system where the design intent is preserved through the implementation.

Then, we compared method stereotypes across the three considered software using the dynamic *fI-fO* distribution *m*. The results are shown in Table 6 for a selection of method stereotypes with a large number of representative methods in the systems. As an illustration of the distributions, Fig. 2b shows the three distributions over dynamic *fI-fO* categories for the *collaborator* stereotype. The calculated measure values are mostly below L_1, with the exception of the *controller* stereotype, and the *get* stereotype in the case of the muCommander. This indicates that most of the method stereotypes are implemented consistently across the three software systems, but also that some *controller* methods may need revision and re-engineering in those systems, as well as some of *get* methods in the muCommander.

In summary, the results indicate that the behavior of different method stereotypes in each of the considered systems is different, and that the behavior of the same method stereotype across the three systems is similar. These results suggest that the three software systems are consistent in terms of designed and implemented behavior. Our results also

TABLE 5. MEASURED DIFFERENCES FOR METHOD PROTOTYPES WITHIN ARGOUML

	GET	PROPERTY	SET	SET COLLAB	COMMAND COLLAB	CONSTRUCT	FACOTRY COLLAB	COLLABORATOR	CONTROLLER	LOCAL CTRL	INITIALIZER
GET	0	1.583	1.164	2.437	2.538	1.869	1.824	1.992	1.791	1.836	0.859
PROPERTY	0.884	0	0.262	0.301	0.553	0.297	1.196	0.532	0.623	0.545	0.553
SET	0.601	0.461	0	1.512	1.615	0.638	1.441	0.609	1.213	0.607	0.029
SET COLLAB	1.344	0.945	0.807	0	0.102	0.496	0.234	0.114	0.926	0.873	1.546
COMMAND COLLAB	1.849	0.504	0.949	0.062	0	0.407	0.398	0.125	0.998	0.673	1.030
CONSTRUCT	1.810	1.185	0.624	0.732	0.846	0	0.538	0.306	0.852	0.483	0.386
FACTORY COLLAB	2.089	1.674	1.151	0.374	0.583	0.562	0	0.212	0.939	0.879	1.074
COLLABORATOR	1.214	0.737	0.573	0.449	0.645	0.295	0.234	0	0.762	0.522	0.339
CONTROLLER	0.034	0.280	0.219	1.071	1.179	0.341	0.177	0.774	0	0.458	0.522
LOCAL CTRL	1.742	0.759	0.446	0.302	0.156	0.221	0.318	0.031	0.233	0	0.488
INITIALIZER	1.208	0.270	0.158	0.229	0.115	0.425	1.179	0.853	0.864	0.547	0

TABLE 6. MEASURED DIFFERENCES FOR METHOD STEREOTYPES ACROSS THE SOFTWARE SYSTEMS

Software vs software	Measured difference
COLLABORATOR	
ArgoUML vs muCommander	0.10024
muCommandervs ArgoUML	0.08285
JabRef vs muCommander	0.17379
muCommandervs JabRef	0.12294
ArgoUML vs JabRef	0.16617
JabRef vs ArgoUML	0.14094
CONSTRUCTOR	
ArgoUML vs muCommander	0.23624
muCommandervs ArgoUML	0.30279
JabRef vs muCommander	0.12724
muCommandervs JabRef	0.26353
ArgoUML vs JabRef	0.21568
JabRef vs ArgoUML	0.23823
INITIALIZER	
ArgoUML vs muCommander	0.37862
muCommandervs ArgoUML	0.18876
JabRef vs muCommander	0.31543
muCommandervs JabRef	0.20829
ArgoUML vs JabRef	0.15565
JabRef vs ArgoUML	0.52043

Software vs software	Measured difference
GET	
ArgoUML vs muCommander	1.07147
muCommandervs ArgoUML	0.51303
JabRef vs muCommander	2.13944
muCommandervs JabRef	0.76360
ArgoUML vs JabRef	0.10767
JabRef vs ArgoUML	0.13943
SET	
ArgoUML vs muCommander	0.18210
muCommandervs ArgoUML	0.06373
JabRef vs muCommander	0.08430
muCommandervs JabRef	0.07264
ArgoUML vs JabRef	0.19017
JabRef vs ArgoUML	0.02758
CONTROLLER	
ArgoUML vs muCommander	1.39183
muCommandervs ArgoUML	2.42161
JabRef vs muCommander	0.61756
muCommandervs JabRef	5.10724
ArgoUML vs JabRef	1.46486
JabRef vs ArgoUML	0.93945

suggest that the dynamic *fI-fO* distribution measure is potentially useful for assessing the implementation of software systems in relation with their design intent.

D. Threats to validity

As with any case studies, any generalization is likely to be limited and should be done carefully. In addition, due to the data available to us, some of the results should be interpreted with care. Specifically, the available traces for the three systems covered only about 50% of the methods, on average i.e. not all methods in the code of the software get executed during the considered run-time executions of the software, and not all methods that may call a given method according to the static analysis of the software, do actually call the given method through the considered execution traces. In terms of conclusion validity, we expect that the results would change to some degree if we used traces that achieve full method coverage. We expect that the differences would not be big enough to invalidate the major conclusions regarding the relationships between method stereotypes and the *fI-fO* categories.

VI. CONCLUSIONS AND FUTURE WORK

We introduced an approach based on a dynamic measure to assess and quantify the extent to which the design intent expressed by method stereotypes is reflected by the run-time behavior of software systems. In principle, OO methods are expected to consistently behave according to their design intent, i.e., that the implied behavior of the methods matches their actual run-time behavior. However in practice this may not be the case and it is likely that some level of mismatch between the intended and actual run-time behavior of methods exists. Our approach provides a way to measure this mismatch between design and implementation. Large mismatch measures indicate low consistency between the design intent and the behavior of a system, while low mismatch indicates high consistency. A qualitative assessment of software systems based on the *fI-fO* measure can support software maintenance activities. We showed that it is possible to identify, for instance, method stereotypes and particular methods that contribute to the increase of the measured mismatch. This will allow the focusing of the re-engineering effort on such identified methods.

The proposed approach can be extended by considering multi-step call traces instead of the single step fan-in and fan-out method calls. Naturally, this would make the analysis more complicated, but could potentially reveal finer-grained aspects of the implementation of the design intentions. We plan to apply our approach to a broader set of software systems to exemplify the design and implementation advice that can be derived from our *fI-fO*-based assessment approach. We also intend to investigate the consideration of multi-step call traces to assess the implementation of design intentions.

REFERENCES

[1] M. S. Feather, S. F. Fickas, A. van Lamsweerde, and C. Ronsard, "Reconciling system requirements and runtime behavior," in *9th International Workshop on Software Specification and Design (IWSSD'98)*, Ise-Shima, 1998, pp. 50-59.

[2] R. R. Lutz, "Software engineering for safety: a roadmap," in *Conference on The Future of Software Engineering (FoSE'00)*, Limerick, Ireland, 2000, pp. 213-226.

[3] N. Dragan, M. L. Collard, and J. I. Maletic, "Reverse Engineering Method Stereotypes," in *22nd IEEE International Conference on Software Maintenance (ICSM'06)*, 2006, pp. 24 - 34.

[4] G. Antoniol, R. Fiutem, and L. Cristoforetti, "Using Metrics to Identify Design Patterns in Object-Oriented Software," in *5th IEEE International Symposium on Software Metrics (METRICS'98)*, Bethesda, MD, USA, 1998, pp. 23-24.

[5] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. T. Halkidis, "Design Pattern Detection Using Similarity Scoring," *IEEE Transactions on Software Engineering,* vol. 32, pp. 896-909, November 2006.

[6] J. K.-Y. Ng, Y.-G. Gueheneuc, and G. Antoniol, "Identification of Behavioural and Creational Design Motifs through Dynamic Analysis," *Journal of Software Maintenance and Evolution: Research and Practice,* vol. 22, pp. 597-527, December 2010.

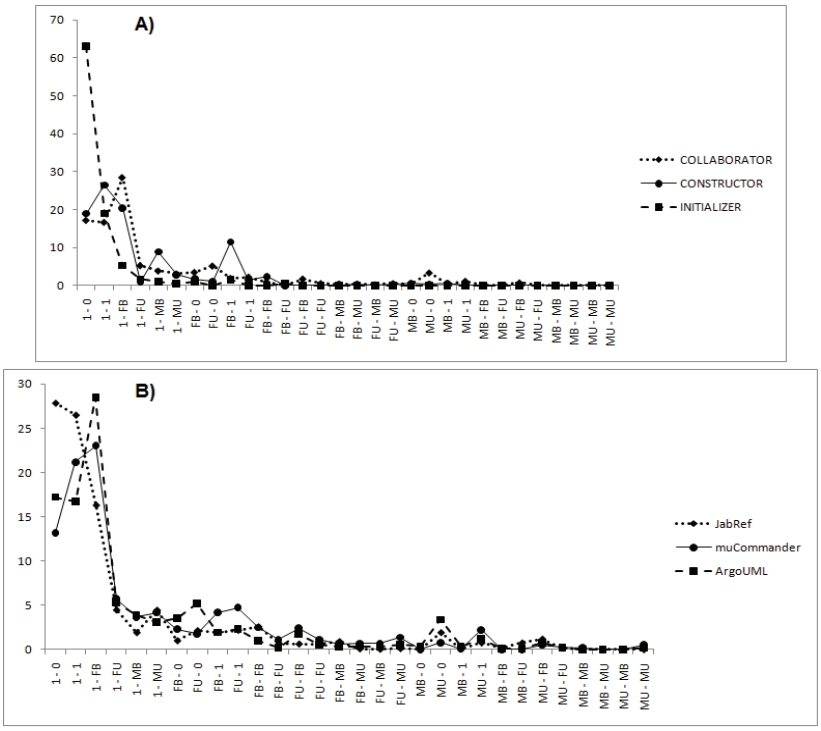

Fig. 2. Distributions over dynamic *fI-fO* categories for (a) the *collaborator*, *constructor* and *initializer* stereotypes in the ArgoUML; and (b) the *collaborator* stereotype in the three software systems. In both cases, the vertical axis shows the percentage of methods belonging to the dynamic *fI-fO* categories, shown on the horizontal axis.

[7] D. Heuzeroth, T. Holl, G. Högström, and W. Löwe, "Automatic Design Pattern Detection," presented at the 11thh IEE International Workshop on Program Comprehension, 2003.

[8] J. Gil and I. Maman, "Micro patterns in Java code," presented at the 20th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, San Diego, CA, USA, 2005.

[9] J. Singer, G. Brown, M. Luján, A. Pockok, and P. Yiapanis, "Fundamental Nano-Patterns to Characterize and Classify Java Methods," *Electronic Notes in Theoretical Computer Science (ENTCS),* vol. 253, pp. 191-204, September 2010.

[10] N. Dragan, M. L. Collard, and J. I. Maletic, "Automatic Identification of Class Stereotypes," in *26th IEEE International Conference on Software Maintenance (ICSM'10)*, Timisoara, Romania, 2010, pp. 1 -10.

[11] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koshcke, "A Systematic Survey of Program Comprehension through Dynamic Analysis," *IEEE Transactions on Software Engineering,* vol. 35, pp. 684-702, September-October 2009.

[12] B. Korel, "Program Slicing in Understanding of Large Programs " presented at the 6th Inernational Workshop on Program Comprehension, Ischia, 1998.

[13] J. Bohnet and J. Dollner, "Visual Exploration of Function Call Graphs for Feature Location in Complex Software Systems," presented at the ACM Symposium on Software visualization (SoftVis'06), New York, NY, USA, 2006.

[14] S. R. Chidamber and C. F. Kemerer, "A Metrics Suite for Object Oriented Design," *IEEE Transactions On Software Engineering,* vol. 20, pp. 476-493, June 1994.

[15] s. M. Yacoub, H. H. Ammar, and T. Robinson, "Dynamic Metrics for Object Oriented Designs," in *6th International Symposium on Software Metrics (METRICS'99)*, Boca Raton, FL, USA, 1999, pp. 50-61.

[16] J. K. Chhabra and V. Gupta, "A survey of dynamic software metrics," *Journal of Computer Science and Technology,* vol. 25, pp. 1016-1029, September 2010.

[17] A. Tahir and S. G. MacDonell, "A systematic mapping study on dynamic metrics and software quality," in *28th IEEE International Conference on Software Maintenance (ICSM'12)*, Trento, Italy, 2012, pp. 326-335.

[18] E. Arisholm, L. C. Briand, and A. Foyen, "Dynamic Coupling Measurement for Object-Oriented Software," *IEEE Transactions On Software Engineering,* vol. 30, pp. 491-506, August 2004.

[19] A. Mitchell and J. F. Power, "Using object-level run-time metrics to study coupling between objects," in *ACM Symposium on Applied Computing (SAC'05)*, Santa Fe, NM, 2008, pp. 1456-1462.

[20] S. Counsell, S. Swift, and J. Crampton, "The interpretation and utility of three cohesion metrics for object-oriented design," *ACM Transactions on Software Engineering and Methodology (TOSEM),* vol. 15, pp. 123-149, 2006.

[21] F. Brito e Abreu and W. L. Melo, "Evaluating the Impact of Object-Oriented Design on Software Quality," in *3rd International Symposium on Software Metrics: From Measurement to Empirical Results*, Berlin, Germany, 1996, pp. 90-99.

[22] L. Briand, J. Wust, J. Daly, and V. Porter, "Exploring the relationship between design measures and software quality in object-oriented systems," *Journal of System and Software,* vol. 51, pp. 245-273, May 2000.

[23] B. Kitchenham, "What's up with software metrics? - A preliminary mapping study," *Journal of Systems and Software,* vol. 83, pp. 37-51, January 2010.

[24] N. Moha, Y.-G. Gueheneuc, A.-F. Le Meur, L. Duchien, and A. Tiberghien, "From a domain analysis to the specification and detection of code and design smells," *Journal of Formal Aspects of Computing,* vol. 22, pp. 345-361, May 2010.

[25] D. Garlan, R. Allen, and J. Ockerbloom, "Architectural Mismatch: Why Reuse Is So Hard," *IEEE Software,* vol. 12, pp. 17-26, November 1995.

[26] D. Garlan, R. Allen, and J. Ockerbloom, "Architectural Mismatch: Why Reuse Is Still So Hard," *IEE Software,* vol. 26, pp. 66-69, July-August 2009.

[27] L. Moreno and A. Marcus, "JStereoCode: Automatically Identifying Method and Class Stereotypes in Java Code," in *27th IEEE/ACM International Conference on Automated Software Engineering (ASE'12)*, Essen, Germany, 2012, pp. 358-361.

[28] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature Location in Source code: A Taxonomy and Survey," *Journal of Software Maintenance and Evolution: Research and Practice,* vol. 25, pp.53-95, January 2011.