How We Design Interfaces, and How To Assess It

Hani Abdeen, Houari Sahraoui, Shata Osama

How We Design Interfaces, and How To Assess It

Hani Abdeen*, Houari Sahraoui† and Osama Shata*
* Department of Computer Science Engineering, Qatar University, Qatar
hani.abdeen@qu.edu.qa – sosama@qu.edu.qa
† DIRO, Université de Montréal, Montréal(QC), Canada
sahraoui@iro.umontreal.ca

***Abstract*—Interfaces are widely used in Java applications as central design elements for modular programming to increase program reusability and to ease maintainability of software systems. Despite the importance of interfaces and a considerable research effort that has investigated code quality and concrete classes' design, few works have investigated interfaces' design. In this paper, we empirically study interfaces' design and its impact on the design quality of implementing classes (i.e., class cohesion) analyzing twelve Java object-oriented applications. In this study we propose the "Interface-Implementations Model" that we use to adapt class cohesion metrics to assess the cohesion of interfaces based on their implementations. Moreover, we use other metrics that evaluate the conformance of interfaces to the well-known design principles "Program to an Interface, not an implementation" and "Interface Segregation Principle". The results show that software developers abide well by the interface design principles cited above, but they neglect the cohesion property. The results also show that such design practices of interfaces lead to a degraded cohesion of implementing classes, where these latter would be characterized by a worse cohesion than other classes.**

***Keywords*-Object-Oriented; Interfaces; Interface Design Principles; Program to an Interface Principle; Interface Segregation Principle; Cohesion; Empirical Study**

I. Introduction

Nowadays, interfaces are frequently used as central design elements in object-oriented applications [1], [2], [3]. Interfaces are meant to govern interactions between semi-independent software classes and to encode shared similarities between different class-types [4], [1]. Good design and use of interfaces can significantly ease program comprehension and software maintainability by fostering modularization and minimizing the impact caused by changes in the software implementation [5], [1], [3]. However, designing good interfaces is a sensitive task with a large influence on the rest of the system [5], [2].

Most existing research efforts largely investigated the design quality and detection of design anomalies at the class level without focusing on the specifics of interfaces [6], [7], [8]. This paper aims to address this gap as it constitutes a potential barrier towards fully realizing the benefits of using interfaces as a central element for modular design.

There are two important principles for interface design, which are: *"Program to an interface, not an implementation"* principle [5, GoF], that we refer to it by PTIP ("Program to Interface Principle"), and the *"Interface Segregation Principle (ISP)* [2]. The rational behind the PTIP is to increase the reusability and flexibility of programs by reducing dependencies upon implementations. The rational behind the ISP is to avoid 'fat' interfaces serving diverse clients, so that clients should not depend upon interface's methods that they do not actually use.

From another perspective, an interface implies the definition and implementation of all its declared methods in their implementing classes. Naturally, those implementing classes would have different semantics and responsibilities, and may be implementing, at the same time, a variety of interfaces [9], [5], [10]. Still, those classes should be cohesive as much as possible [6], [11], [12], [13]. Studying empirically the design quality of interfaces, assessing their compliance to well-known interface design principles, and/or examining the relations between the design quality of interfaces and that of their implementing classes remain important research challenges.

Recent studies showed that interfaces which do not respect the ISP (as measured by the Service Interface Usage Cohesion (SIUC) metric [14]) are more likely to be change-prone interfaces than those respecting ISP [3]. Yet, we still do not know if those change-prone interfaces do not abide the ISP because they adhere to another design principle, or just because they are badly designed as 'fat' interfaces. In other words, it is important to determine if ISP, PTIP, and (interface/implementing class) cohesion are conflicting design properties, or if they can be achieved jointly. Finally, despite many claims about those cited design principles, to the best of the authors' knowledge, there is no reported study about *how do software developers design and use interfaces?*: e.g., do really developers try to abide by those principles?
We believe addressing those questions can give important support to assist maintainers and designers in their decisions about interface design.

Contributions

To understand how programmers design and use interfaces, we conduct an empirical study, covering 12 Java real-world applications, that aims at assessing interface design with regard to different properties, namely PTIP, ISP and Cohesion. To assess the compliance of interfaces to the ISP and PTIP principles, we reuse the Service Interface Usage Cohesion (SIUC) metric [14] for measuring ISP, and we develop a new metric, namely Loose Program to Interface (LPTI) for measuring PTIP. The impact of interface design on the cohesion of implementing classes is studied based on the "Interface-Implementations Model" (IIM). This model allows us to adapt existing class cohesion metrics for assessing the internal cohesiveness of interfaces based on their implementing classes. The adapted metrics are the Tight (and Loose) Class Cohesion metrics TCC (and LCC) [11], [12], and the Sensitive Class Cohesion Metric (SCOM) [15], [13]. Using the above-mentioned metrics and 12 open-source Java applications, we conduct statistical analyses to address the following research questions:

Q1 Do software developers abide by the design properties PTIP, ISP and Cohesion? we examine empirically whether interfaces in real-world applications abide by the design principles, PTIP, ISP and Cohesion.

Q2 Is there any conflict between the interface design properties? we examine the correlations between the ISP and PTIP, and also between those design principles and the internal cohesiveness of interfaces.

Q3 Do interfaces threaten implementations' cohesion? we examine the correlation between the cohesion at the interface level and that at the level of implementing classes. Then we compare between the cohesion of classes implementing interfaces and the cohesion of classes which do not implement interfaces.

The remainder of the paper is organized as follows. Section II presents the background and vocabularies we use in this paper. Section III discusses relevant existing work related to interface design. We detail our metrics and the empirical study, with the research hypotheses, respectively in Section IV and Section V. Section VI presents and analyses the results of the empirical study, and describes the findings. Finally, we present threats to their validity in Section VII before concluding.

II. Background and Vocabularies

Before detailing our study for analyzing design quality attributes of interfaces, we introduce the vocabularies and notations we use in the rest of this paper.

Interface: we consider an interface as a set of method declarations (i.e., a set of signatures). In this paper, we do not take into account "abstract classes", "marker interfaces", interfaces declaring only constants, nor interfaces used only by unit tests ("moke interfaces"). We define the size of an interface i, $size(i)$, by the number of methods that i declares.

Interface Implementations: a class c is defined as an implementing class to an interface i if it defines some *not-abstract* (concrete) method(s) overriding some method(s) of the i's methods. For example, if a not-abstract class c implements explicitly an interface i_1 (i.e., `class` c `implements` i_1), and i_1 inherits from another interface i_2 (i.e., `interface` i_1 `extends` i_2), then c is an implementing class of both interfaces i_1 and i_2 –since c must override and implement all the methods declared in i_1 and i_2. If another class c_sub extends c and defines not-abstract methods overriding some methods of i_1 and i_2, but not necessarily all their methods, then c_sub is also an implementing class of i_1 and i_2 –even though c_sub does not explicitly implement i_1 and i_2.

However, a special subset of the implementing classes of an interface is the *Direct-implementations*, which are the classes that *explicitly* implement the considered interface. In the example above, the class c is a direct implementation of i_1, but it is not a direct implementation of i_2.

We use $Imp\_C(i)$ to denote the set of all implementing classes of an interface i. We also use $Imp\_M(i)$ to denote the set of all the concrete methods that override i's methods, which we denote by $M(i)$. We denote the subset of direct implementing classes of i by $DImp\_C(i)$.

Program to Interface Principle (PTIP): the idea behind the PTIP, as mentioned by GoF [5], is to reduce coupling upon implementations and to achieve flexibility in swapping used objects with different implementations. To this end, interfaces (or abstract base classes) should be used as reference types instead of their implementing classes whenever appropriate [1]. Venners [1] explains in detail and by example how programming to interfaces can add more extensibility and flexibility to OO programs than programming to abstract classes. Nevertheless, Gamma outlines the fact: "As always there is a trade-off, an interface gives you freedom with regard to the base class, an abstract class gives you the freedom to add new methods later. ..."[1] However, it is worth noting that this paper is limited to the investigation of interface design, not abstract classes. Whether it is better to use abstract classes rather than interfaces is out of the scope of this paper.

With regard to the PTIP, wherever an interface, or a class, x is used as a type (e.g., a type for an instance variable) we say that there is a *reference* pointing to x. We denote the set of all references pointing to an interface, or a class, x by $Ref(x)$.

Interface Segregation Principle (ISP): interface changes may break the interface clients [3]. Hence, clients should not be forced to depend upon interfaces' methods that they do not actually use. This observation leads to the ISP principle which states that an interface should be designed 'as small as possible', so as it declares only methods that are all used by the interface's clients [2].

With regard to the ISP, we use the same definition of interface clients as [3]: we define a *class* c as a client to an interface i if some method in c calls *any* of the methods overriding the i's methods, and/or calls *directly* any of the interface methods –using the polymorphism mechanism. In this definition, we do not care which concrete method will be invoked at runtime, whilst we know that the invoked method will be surely an implementation method of the concerned interface. Hence, the class stating the invocation is a client to that interface.

III. Related Work

Existing research efforts that are related to object-oriented software design quality mainly revolve around code quality, code smells, and refactoring support for classes (implementations). Few recent articles attempt to address the particularities of interface design.

A. Source Code Metrics

In literature, there is a large list of source code metrics aiming at assessing the design of OO programs, particularly at class level [6], [16], [11], [12]. The most known ones are the object-oriented metrics by Chidamber and Kemerer (CK) [6]. Chidamber and Kemerer proposed the first class cohesion metric, Lack of Class Cohesion Metric (LCOM), for measuring the lack of class cohesion. This metric has been widely reinterpreted and revised: LCOM2 [6], LCOM3 [17], LCOM4 [18] and LCOM5 (or LCOM*) [16]. Studies report two problems about those metrics [12], [13], [15]: (1) all of LCOM1 ... LCOM4 do not have an upper limit, hence it is hard to interpret their computed values; (2) all those metrics evaluate the lack of class cohesion instead of measuring the class cohesion itself.

To evaluate existing metrics for class cohesion, on the one hand, Etzkorn et al [12] compares various OO class cohesion metrics (including the LCOMs) with ratings of experts to determine which of these metrics best match human-oriented views of cohesion. The study concludes that the LCC metric (Loose Class Cohesion), as defined by Bieman *et al.* [11], is the metric which best matches the expert opinion of class cohesion. The study also shows that LCC has a strong correlation with the Tight Class Cohesion (TCC), also proposed by Bieman *et al.* in [11]. On the other hand, recently, Al-Dallal

[1] A conversation between Bill Venners and Erich Gamma [5] (June 6, 2005): www.artima.com/lejava/articles/designprinciples.html

[13] performed a study investigating the discriminative power of 17 class cohesion metrics, including the LCOMs, LCC and TCC metrics. The discriminative power of a cohesion metric is defined as the probability to produce distinct cohesion values for classes with the same number of attributes and methods but different connectivity pattern of cohesive interactions (CPCI) [13]. This is important since a highly discriminating cohesion metric exhibits a lower chance of incorrectly considering classes to be cohesively equal when they have different CPCIs. The study results show that the Sensitive Class Cohesion Metric (SCOM), as defined by Fernández and Peña in [15], has the best discriminative power.

Although those cohesion metrics are valuable, they unfortunately do not address the particularities of interfaces [3] –since interfaces do not contain any logic, such as method implementations, invocations, or attributes. To the best of our knowledge, there is no reported study mapping those metrics for evaluating the cohesion at the interface level, and/or investigating the relation between the cohesion of implementing classes and interfaces' design.

B. Refactoring Support

In recent years, there has been considerable interest in automatic detection and correction of design defects in object oriented software [19], [20], [21]. Mens and Tourwé [8] survey shows that existing approaches are mainly based on source code metrics and predefined bad smells in source code, except for a very few contributions such as [22].

Fowler and Beck [7] propose a set of bad smells in OO class design: e.g., Data class, Blob class, Spaghetti code. Based on Fowler and Beck's definitions of class smells, several approaches to automatically improving code quality have been developed. Abbes *et al.* [23] performed an empirical study on the impact of Blob and Spaghetti antipatterns on program comprehension. The results conclude that the combination of those antipatterns have a significant negative impact on system comprehensibility. Liu *et al.* [19] provide a deep analysis of the relationships among different kinds of bad smells and their influence on resolution sequences.

Unfortunately, none of those code smells and OO metrics are applicable to Software interfaces. In the literature, few recent articles attempt to address design defects of interfaces.

The authors in [24] study, through 9 open-source OO applications, the correlation between the presence of method clones in different interfaces and the presence of code clones in implementation methods. The results show that there is always a positive correlation, even though moderate, between the presence of method clones in interfaces and the presence of code clones in implementation methods.

Recently, Mihancea and Marinescu [9] studied several recurrent patterns of using inheritance and polymorphism that can impact program comprehensibility (comprehension pitfalls). One of their comprehension pitfalls is the "Partial Typing", which outlines that interfaces cannot be always assumed as complete types of all their implementing classes. Precisely, it outlines that the variability in the semantics and behaviors of implementing classes of an interface can be very strong in some situations. However, despite the importance of this work for detecting and avoiding the comprehension pitfalls, this work does not investigate interface design and/or its impact on the internal quality of classes (i.e., cohesion).

C. Interface Metrics

Boxall and Araban define a set of counter metrics to measure the complexity and usage of interfaces [25]. Their metrics return the number of interface methods, all arguments of the interface methods, the interface client classes, etc. The authors in [26] define more complex metrics that assess the interface design quality with regard to existing similarities among interfaces, and with regard to the redundancy in interface sub-class hierarchies.

As outlined in the introduction and Section II, with regard to the ISP, Romano and Pinzger [3] used the Service Interface Usage Cohesion (SIUC), as defined by Perepletchikov [14], to measure the violation of the ISP. Romano and Pinzger refer to SIUC by Interface Usage Cohesion metric (IUC). Their work concludes that in order to limit the impact of interface changes and facilitate software maintenance, interfaces should respect the ISP: i.e., interfaces should be characterized by high values of SIUC metric.

However, despite the success of this cited body of research efforts on interface design metrics, to the best of the authors' knowledge, up to date, there is no reported study about the compliance of interface design in real-world OO applications to the ISP (and/or PTIP and Cohesion).

D. Interface Design

Romano and Pinzger [3] investigated the suitability of existing source code metrics (CK metrics, interface complexity and usage metrics, and the SIUC metric) to classify interfaces into change-prone and not change-prone. The paper concluded that most of the CK metrics are not sound for interfaces. Therefore this confirms the claim that interfaces need to be treated separately. The SIUC metric exhibits the strongest correlation with the number of interface changes. Hence, they conclude that the SIUC metric can improve the performance of prediction models for classifying interfaces into change-prone and not change-prone.

The authors in [27] report a technique and tool support to automatically detect patterns of poor API usage in software projects by identifying client code needlessly re-implementing the behavior of an API method. Their approach uses code similarity detection techniques to detect cases of API method imitations then suggest fixes to remove code duplication.

The work in [28] examined the relative impact of interface complexity (e.g. interface size and operation argument complexity) on the failure proneness of the implementation using data from two large-scale systems. This work provides empirical evidence that the increased complexity of interfaces is associated with the increased failure proneness of the implementation (e.g., likelihood of source code files being associated with a defect) and higher maintenance time.

This present paper goes further by studying whether software developers abide by the design principles of interfaces and/or the cohesion quality of implementing classes. Moreover, we empirically study the correlations among those different design properties of interfaces, and study the impact of interface design on the cohesion of implementing classes.

IV. Assessing Interface Design

We are interested in assessing interface design from two perspectives, conformance to design principles and cohesion. To this end, we define, reuse, and adapt a set of metrics to capture these properties.

A. Metrics For Assessing Interface Design Quality

This section describes the interface metrics that we use to assess interface design with regard to the interface design principles PTIP and ISP.

Similarly to the work by Romano and Pinzger in [3] (see Section III-C), we also use the Service Interface Usage Cohesion (SIUC) to measure the compliance of an interface to the ISP. SIUC is defined by Perepletchikov [14] as follows:

$$SIUC(i) = \frac{\sum_c \frac{num\_used\_mtds(i,c)}{size(i)}}{\mid Clients(i) \mid} \quad \forall c \in Clients(i) \quad (1)$$

Where $num\_used\_mtds(i, c)$ is the number of i's methods that are used by the client class c; and $Clients(i)$ is the set of all client classes to i. SIUC states that an interface has a strong cohesion if each of its client classes actually uses all the methods declared in the interface. SIUC takes its value in the interval [0..1]. The larger the value of SIUC, the better the interface usage cohesion is.

As for the PTIP, to the best of our knowledge, up to date, there is no reported study proposing a measurement or a tool for assessing the adherence of interface design to the PTIP. Following the description of the PTIP in Section II, the PTIP can be interpreted as follows: instead of using a specific concrete class as a reference type, it would achieve more flexibility to use the interface declaring the methods required to perform the desired service [5], [1], [4].

Thus, to assess the design property PTIP, we need to check the dependencies pointing to specific implementing classes rather than to their interfaces. Let $Ref(i)$ denotes the set of all references pointing to the interface i, and let $Ref\_DImp(i)$ denotes the set of all references pointing to the 'direct' implementations of i (see Section II). We define the Loose Program To Interface metric (LPTI)[2] as the normalized ratio $Ref(i)$ by $Ref\_DImp(i)$:

$$LPTI(i) = \frac{\mid Ref(i) \mid}{\mid Ref(i) \cup Ref\_DImp(i) \mid} \quad (2)$$

LPTI takes its values in [0..1], where the larger the LPTI value, the better is the interface adherence to the PTIP.

B. Selecting Cohesion Metrics

As mentioned earlier, cohesion is one of the properties we consider when assessing the interface design. Rather than defining new interface cohesion metrics, we decided to select class cohesion ones and adapt them. This eases the study of the relationship between interface and class cohesion. We set the following criteria for selecting the cohesion metrics for our study: (1) selected cohesion metrics should correlate very well with the human-oriented view of class cohesion [12]; (2) they also should have a good discriminative power [13]; (3) finally, to easily interpret metrics values, it is preferable to select metrics that measure class cohesion instead of lack of cohesion, and take their values in a specific range (e.g., [0..1]).

Following the discussion presented in Section III-A about class cohesion metrics, none of the LCOMs metrics satisfies our 3rd criterion. According to the study by Etzkorn *et al.* [12], the LCC metric best matches the human-oriented view of class cohesion. LCC metric, as defined by Bieman and Kang [11], measures class cohesion and takes its values in [0..1]. Thus, LCC satisfies our 1st and 3rd criterion. However, according to the study by Al-Dallal [13], the metric which has the best discrimination power is the SCOM metric as defined in [15]. SCOM metric measures class cohesion and takes its values in [0..1]. Thus, SCOM satisfies our 2nd and 3rd criteria. Nevertheless, on the one hand, there is no reported study about the correlation between SCOM and human-oriented view about class cohesion. On the other hand, the study of Al-Dallal shows that LCC does not have a good discriminative power; but the TCC metric, which is also defined by Bieman and Kang [11] and has a strong correlation with LCC [12], has a better discriminative power. According to all this above, we select all of SCOM, LCC and TCC metrics for assessing class cohesion in our experiments. Table I describes these selected metrics.

[2]The name #"Loose" Program To Interface# outlines that this metric does not consider references pointing to indirect implementations (Section II).

Table I
DESCRIPTIONS FOR SELECTED CLASS COHESION METRICS.

DEF1. Attributes	The attributes of a class c, $A(c)$, are all attributes that are defined in c in addition to the not-private attributes defined in the super classes of c (i.e., inherited ones).
DEF2. Local Invocation	We say that there is a local invocation going from a method $m1$ to another one $m2$ if in the $m1$'body there is a call to $m2$ via `this`/`self` or `super` keywords (i.e., an invocation within the scope of the same object).
DEF3. Attribute Access	By definition, a method m accesses an attribute a of its class, if a appears in the m's body, or within the body of another method that is invoked by m, directly or transitively via local invocations.
DEF4. Methods Connectivity	Two methods belonging to the same class c are defined as *"directly"* connected if they access at least one shared attribute of c. In the same vein, m_1 and m_n, are said *"indirectly"* connected if there is a sequence of methods m_2, m_3, ..., m_{n-1}, such that: $m_1 \delta m_2$, ..., $m_{n-1} \delta m_n$; where $m_x \delta m_y$ represents a direct connection.
Tight Class Cohesion (**TCC**)*[11]	TCC(c) = the ratio of directly connected pairs of visible methods within c, $DCpM(c)$, by the number of all possible pairs of visible methods in c: $NP(c)$. $$TCC(c) = \frac{\mid DCpM(c) \mid}{NP(c)} \quad (3)$$
Loose Class Cohesion (**LCC**)*[11]	LCC(c) = the ratio of directly or *indirectly* connected pairs of visible methods within c, $CpM(c)$, by $NP(c)$. $$LCC(c) = \frac{\mid CpM(c) \mid}{NP(c)} \quad (4)$$
Sensitive Class Cohesion Metric (**SCOM**)*[15]	SCOM(c) = the normalized ratio of the summation of the weighted connection intensity between all pairs of methods in c. The weighted connection intensity between two methods x and y is defined as follows: $Con_{x,y} = a_{x,y} \times (\mid A(x) \cap A(y) \mid / min(\mid A(x) \mid, \mid A(y) \mid))$. Where $a_{x,y} = \mid A(x) \cup A(y) \mid / \mid A(c) \mid$, and $A(x)$ and $A(y)$ are respectively the sets of c's attributes that are accessed by x and y, and $A(c)$ is the set of all c's attributes. $$SCOM(c) = \frac{\sum_q \sum_p Con_{q,p}}{NP(c)} \; \forall m_q, m_p \in M(c) \quad (5)$$

* All these metrics are applicable only if $\mid M(c) \mid > 1$. They take their values in [0..1], where 0 is the worst value and 1 is the ideal one.

C. Adapting Cohesion Metrics

Recall that it is not possible to evaluate interface cohesion without additional heuristics –since interfaces do not contain any logic such as attributes and/or method implementations. To evaluate interface cohesion, our approach retrieves information at the interface implementations to capture coupling between the interface methods, and assess the cohesion at the interface level. Therefore, we propose and use the Interface-Implementations Model (IIM).

In the IIM of a software project p, IIM(p), every Interface i is mapped to an entity namely *Integral-Interface*, that we refer to by $\int i$. An $\int i$ consists of a list of methods, $M(\int i)$, and a list of attributes, $A(\int i)$, so that $\int i$ can be thought as a class with methods, attributes and accesses. Each method in $\int i$, $m_{\int i}$, represents its corresponding method in i, m_i, with all its associated implementation methods $Imp\_M(m_i)$. Thus, a $m_{\int i}$ can be thought as a composition of all the implementation methods of m_i. Hence, we say that a $m_{\int i}$ accesses an attribute a if any of the implementation methods of m_i accesses a. Using $A(mx)$ to denote the set of attributes accessed by mx, the set of attributes accessed by a method $m_{\int i}$, and the set of all attributes that are associated to $\int i$, $A(\int i)$, are defined as follows: $A(m_{\int i}) = \cup A(m) \quad \forall m \in Imp\_M(m_i)$; and $A(\int i) = \cup A(m_{\int i}) \quad \forall m_{\int i} \in M(\int i)$.

Figure 1 shows an example describing the mapping of an interface ($I1$) to its integral-interface presentation ($\int I1$). Note that some methods, such as `C3.bar()`, access some attributes because of local invocations: `C3.bar()` accesses, in addition to `a3`, the attribute `a1` because it locally invokes `C1.bar()` (via `super`), and this latter accesses `a1` in its body. Similarly, we say that `C2.foo()` accesses, in addition to the attribute `d2`, the attributes `a2` and `b2` (see Table I).

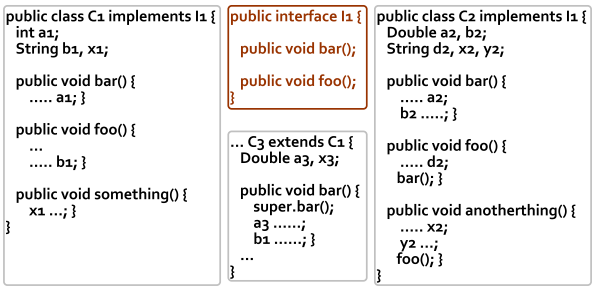
To compute cohesion metrics for interfaces, we use the IIM in which each interface is represented by an integral-interface (see Figure 1). For example, the values of TCC and LCC for the interface `I1` in Figure 1 are maximal: TCC = LCC = 1 –since `I1` consists of only one pair of methods (`bar()`, `foo()`), NP(`I1`) = 1, and these methods are connected: $A(bar()) \cap A(foo()) = \{$`b1, a2, b2`$\}$. The SCOM value for `I1` is: $\frac{(6/6)\times(3/4)}{1} = \frac{3}{4} = 0.75$ –Equation (5).

Note. An integral-interface $\int i$ does not replicate all the information in its implementing classes –since these latter can completely differ in their semantics and behaviors [9], [5], [10], and also define additional methods to those declared in i. Taking for example the implementing classes `C1` and `C2` of the interface `I1` (Figure 1). On the one hand, the values of TCC, LCC for `C2` are maximal: TCC(`C2`) = LCC(`C2`) = 1 –since all the methods in `C2` are connected. Similarly, SCOM(`C2`)= $^{(26/10)}/_{3} = {}^{13}/_{15} \approx 0.87$. On the other hand, the values of TCC, LCC and SCOM for `C1` are minimal: TCC(`C1`) = LCC(`C1`) = SCOM(`C1`) = 0 –since there is no pair of connected methods in `C1`. Recall that the values of TCC, LCC and SCOM for the interface `I1` are respectively 1, 1 and 0.75. Hence, we need to empirically test the correlation between the cohesion at interfaces' level and the cohesion at the level of implementing classes.

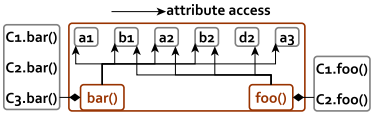
For the implementation, we use the Moose toolkit [29] because its meta-model language, FAMIX, includes descriptions of all software entities (e.g., classes, interfaces, methods, attributes etc.) and their associated relationships (e.g., invocations, attributes accesses, references etc.) that are necessary to build the IIMs and compute the metrics. We implemented all the metrics with this platform, utilizing exactly their formulations presented in previous sections.

V. Empirical Study

In this section, we detail the empirical study that we conduct to answer the research questions outlined in the introduction (Q1, Q2 and Q3).



The interface `I1`, which declares `bar()` and `foo()`, is implemented by 3 classes `C1`, `C2` and `C3`. `C3` extends `C1` and re-overrides only `bar()`.



The representation of the interface `I1` in an IIM ($\int I1$). $\int I1$ consists of `bar()` and `foo()`, which access 6 attributes: $A(\int I1) = \{$`a1, b1, a2, b2, d2, a3`$\}$.

Figure 1. Example about IIM.

A. Goal, Perspective and Target

The goal of the study is to analyze, from the perspective of researchers, the design of interfaces in order to: - evaluate the adherence of real-world interfaces to the design properties PTIP, ISP and Cohesion; - investigate the correlation between those different design properties; - and finally, evaluate the impact of interface design on the cohesion of implementing classes. We believe the results of our study are interesting for researchers and also for software maintainers and engineers who want to fully release the benefits of interfaces for modular design of OO systems.

B. Context

The context of the empirical study consists of twelve open-source object-oriented software applications that are all written in Java. Table II shows information about the size of those applications: (1) ArgoUML$_{v0.28.1}$, (2) Hibernate$_{v4.1.4}$, (3) Jboss$_{v6.0.0}$, (4) Jfreechart$_{v1.0.14}$, (5) JHotDraw$_{v7.1}$, (6) Opentaps$_{v1.5.0}$, (7) Plantuml$_{v7935}$, (8) Proguard$_{v4.8}$, (9) Rapidminer$_{v5.0}$, (10) Spring-RCP$_{v1.1.0}$, (11) SweetHome$_{v3.4}$, and (12) Vuze$_{v4700}$. We selected open-source systems so that researchers can replicate our experiments. However, many of those systems are widely used in both academic and industry (e.g., Hibernate, Jboss, Rapidminer and Spring-RCP). We carefully selected those systems to cover a large variety of systems that differ in utility, size, users and developers community. Finally, each of those systems contains a considerable variety of interfaces: small, medium and large interfaces – except Proguard and SweetHome which contain small sets of relatively small interfaces.

For each application, we collected the values of SIUC and LPTI for each interface, and the values of TCC, LCC and SCOM for each interface and class. Using the Anderson-Darling normality test, we found that the values collected for all the metrics are not normally distributed, with $p-values < 0.05$. This means that parametric tests cannot be used in our study.

C. Hypotheses

The empirical study aims at assessing the following hypotheses with regard to interface design.

Table II
INFORMATION ABOUT CASE-STUDY SOFTWARE PROJECTS.

App	[1]Classes	[2]Interfaces	$\sum size(i)$	Imp	DImp
ArgoUML	1793	88	833	358	177
Hibernate	2658	319	2352	745	563
Jboss	3573	458	3350	648	523
Jfreechart	374	45	231	104	83
JHotDraw	396	42	523	159	58
Opentaps	1257	113	1386	214	167
Plantuml	722	77	311	328	205
Proguard	388	18	164	238	240
Rapidminer	1796	99	628	780	228
Spring-RCP	566	95	558	159	100
SweetHome	355	13	131	30	28
Vuze	2539	683	5724	1493	1270
TOTAL	**16417**	**2050**	**16191**	**5256**	**3642**

$\sum size(i)$ represents the number of all methods declared in interfaces; Imp and $DImp$ denote respectively implementing classes and direct implementation ones; Only classes and interfaces defining/declaring more than 1 visible method are considered.
[1] Concrete (not-abstract) classes, without unit test classes. [2] Excluding interfaces outside the system boundary (e.g., Java library interfaces, such as Serializable interface).

H_{0PTIP} There is no significant evidence showing that interfaces' design respects the PTIP. The alternative hypothesis is:
H_{1PTIP} There is a statistical evidence showing that interfaces adhere to the PTIP.

Identically to the above hypotheses, we define the other hypotheses with regard to the other design properties SIP and Cohesion: H_{0SIP} and H_{1SIP}; and H_{0Coh} and H_{1Coh}. We assess each of the previous null hypotheses by comparing the distance between the average value of the corresponding metric(s) (e.g., SIUC for ISP) to the ideal and worst values of that (those) metric(s). Recall that all used metrics take their values in [0..1], where 0 is the worst value and 1 is the ideal one. For this purpose, we divide the interval [0..1] as follows: **[0 BAD *values* 0.4 INDECISIVE *values* 0.6 GOOD *values* 1]**.

$H_{0\rho(PTIP,ISP,Coh)}$ There is no significant correlation between the interface properties PTIP, ISP and Cohesion. The alternative hypothesis is:
$H_{1\rho(PTIP,ISP,Coh)}$ There is a significant *positive* (or *negative*) correlation between the interface properties. Thus, designing interfaces with regard to one of those properties can anticipate enhancing (or degrading) the quality of interface design with regard to the other correlated property/ies.

We assess the previous null hypothesis for each couple of interface properties (i.e., $H_{0\rho(PTIP,ISP)}$, $H_{0\rho(PTIP,Coh)}$ and $H_{0\rho(ISP,Coh)}$) by studying the Spearman correlations between the values of corresponding metrics.

$H_{0Coh(Int,Imp)}$ There is no significant evidence showing that interfaces' design can impact the cohesion of implementing classes. The alternative hypothesis is:
$H_{1Coh(Int,Imp)}$ Interfaces impact the cohesion of classes.

To assess the $H_{0Coh(Int,Imp)}$, first, we study the Spearman correlation between the cohesion of interfaces and that of their implementing classes; second, we compare the cohesion of implementing classes and not-implementing classes using a two-tailed Wilcoxon test.

VI. RESULTS ANALYSIS

This section presents and analyses the results of our study aiming at assessing the research hypotheses.

A. Interface Design with-respect-to ISP, PTIP and Cohesion

Figure 2 shows that interfaces have a strong tendency to be designed according to the PTIP and ISP properties, but also with neglecting the cohesion property. On the one hand, Figure 2(a) shows that for all studied applications, bar only two applications (JFreeChart and Proguard), the median values of LPTI metric are very high and close (or even equal) to the ideal value 1. The figure shows also that for all studied applications, bar only two applications (JHotDraw and SweetHome), the median values of SIUC metric are very high. On the other hand, the Figure 2(a) shows that the median values of cohesion metrics (TCC, LCC and SCOM) are very close to the worst value in 7 applications (ArgoUML, Hibernate, JBoss, Rapidminer, Spring-RCP, SweetHome and Vuze). Only in 3 applications (JFreeChart, Opentaps and Proguard), interfaces tend to have acceptable cohesion values. These results are evidenced in Figure 2(b), where all interfaces are studied together, and in Figures 2(c), 2(d) and 2(e), where random samples of interfaces are studied.

Table III shows, on the first hand, that the mean values of LPTI and SIUC metrics are decisively high ($Mean > 0.6$), but the mean values of cohesion metrics (TCC, LCC and SCOM) are not decisive in many cases ($0.4 \leq Mean \leq 0.6$). Except for the cases of ArgoUML, Hibernate, Jboss, Rapidminder, SweetHome, and Vuze, mean cohesion values are decisively small: $Mean < 0.4$. This is also true when considering all the interfaces together. On the second hand, Table III shows that the standard deviations of metric values are high in almost all cases, thus we can't draw our conclusions by looking only to the mean values. Still, considering the size of our sample (2050 interfaces from 12 applications), and the mean and median values, we are confident that interfaces generally conform to design principles PTIP and ISP, but are not sufficiently cohesive.

This leads us to the following findings:

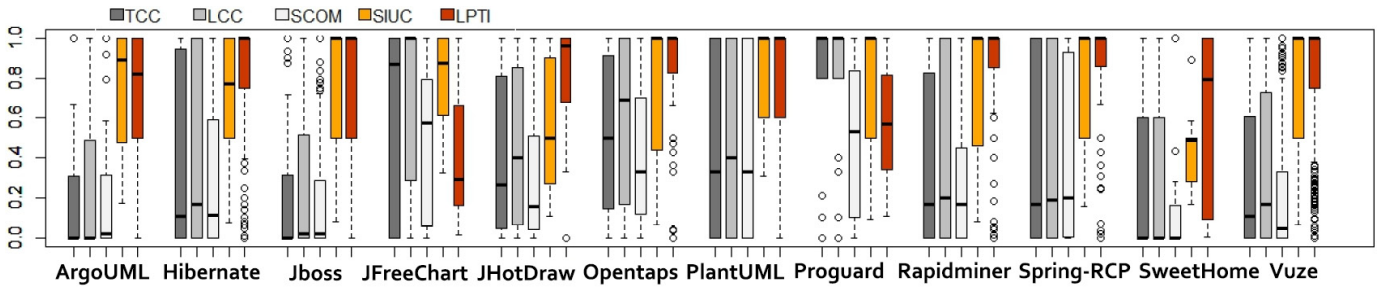
F1 Software developers definitely abide by the *"Program to Interface"* and *"Interface Segregation"* principles.

F2 Developers of interfaces do not abide the *"Cohesion"* property in interfaces' design.

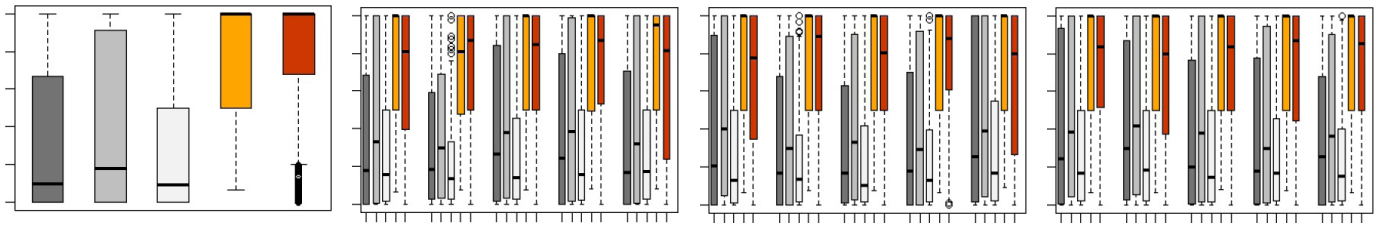
These findings lead us to conjecture that PTIP and ISP, on the one hand, and the Cohesion, on the other hand, could be conflicting interface-design properties.

B. Correlations between ISP, PTIP and Cohesion Properties

To assess the $H_{0\rho(PTIP,ISP)}$, $H_{0\rho(PTIP,Coh)}$ and $H_{0\rho(ISP,Coh)}$ hypotheses, we study the correlations between the involved properties. The results are presented in Table IV. Although the correlations between the interface properties PTIP, ISP, and Cohesion are statistically significant when considering all the applications together, they are not strong enough ($-0.2 \leq \rho \leq 0.11$) to be practically significant. More specifically, correlations between PTIP and ISP are, for all the applications, close to zero except for Rapidminer (-0.45). Besides, there is a weak (to very weak) negative correlation between the PTIP and Cohesion properties in many cases,



(a) All Interfaces per Application



(b) All Interfaces Together (c) First 5 samples (d) Second 5 samples (e) Third 5 samples

Figure 2. Box plots of #TCC, #LCC, #SCOM, #SIUC and #LPTI metrics for interfaces.

Each of 2(c), 2(d) and 2(e), shows box blots of interface metrics for 5 random samples of 100 interfaces taken from all applications and with possible overlaps.

Table III
THE MEAN VALUES AND STANDARD DEVIATIONS (σ) FOR INTERFACE METRICS, BY APPLICATION, AND FOR ALL INTERFACES TOGETHER.

	PTIP		**ISP**		**Cohesion**					
	$\overline{\text{LPTI}}$	σ	$\overline{\text{SIUC}}$	σ	$\overline{\text{TCC}}$	σ	$\overline{\text{LCC}}$	σ	$\overline{\text{SCOM}}$	σ
ArgoUML	**0.70**↑	0.31	**0.74**↑	0.29	*0.20*↓	0.34	*0.26*↓	0.39	*0.25*↓	0.38
Hibernate	**0.83**↑	0.29	**0.71**↑	0.30	*0.36*↓	0.42	0.40	0.44	*0.30*↓	0.38
Jboss	**0.73**↑	0.41	**0.76**↑	0.30	*0.20*↓	0.33	*0.28*↓	0.37	*0.21*↓	0.33
Jfreechart	0.41	0.33	**0.79**↑	0.23	0.58	0.45	**0.65**↑	0.42	0.47	0.39
JHotDraw	**0.82**↑	0.25	0.56	0.33	0.41	0.38	0.46	0.39	*0.31*↓	0.35
Opentaps	**0.85**↑	0.27	**0.74**↑	0.33	0.50	0.38	0.57	0.40	0.42	0.35
PlantUML	**0.76**↑	0.32	**0.83**↑	0.22	0.47	0.44	0.51	0.44	0.56	0.42
Proguard	0.59	0.30	**0.77**↑	0.33	**0.78**↑	0.39	**0.81**↑	0.34	0.50	0.37
Rapidminer	**0.84**↑	0.30	**0.74**↑	0.33	*0.36*↓	0.41	0.40	0.43	*0.29*↓	*0.34*
Spring-RCP	**0.85**↑	0.29	**0.78**↑	0.28	0.40	0.45	0.44	0.46	0.40	0.41
SweetHome	0.60	0.44	0.44	0.22	*0.27*↓	0.39	*0.28*↓	0.39	*0.16*↓	0.29
Vuze	**0.84**↑	0.24	**0.76**↑	0.30	*0.32*↓	0.38	*0.36*↓	0.40	*0.24*↓	0.34
ALL	**0.80**↑	0.31	**0.75**↑	0.30	*0.33*↓	0.40	*0.38*↓	0.41	*0.28*↓	0.36

High Means (> 0.6) and low Means (< 0.4) are respectively in boldface (annotated with ↑) and italic-face (annotated with ↓).

Table IV
SPEARMAN CORRELATION AMONG INTERFACE PROPERTIES ($\alpha = 0.05$).

	PTIP$_{vs.}$**ISP**	**PTIP**(LPTI)$_{vs.}$**Coh.**			**ISP**(SIUC)$_{vs.}$**Coh.**		
	LPTI,SIUC	–,TCC	–,LCC	–,SCOM	–,TCC	–,LCC	–,SCOM
ArgoUML	0.09	0.05	0.07	0.11	-0.30	-0.34	-0.26
Hibernate	-0.19	-0.06	-0.10	-0.02	0.04	0.07	0.17
Jboss	-0.01	-0.22**	-0.25**	-0.22**	0.19	0.16	0.21*
Jfreechart	-0.06	-0.30*	-0.20	-0.16	-0.04	-0.02	0.03
JHotDraw	-0.27	-0.05	-0.07	0.00	0.23	0.09	0.33
Opentaps	0.11	-0.30**	-0.35**	-0.24*	-0.02	0.02	0.17
PlantUML	0.00	0.02	0.00	0.08	0.07	-0.02	-0.06
Proguard	-0.27	**-0.50***	-0.47*	-0.34	**0.76****	**0.76****	0.55
Rapidminer	-0.45**	-0.25*	-0.33**	-0.19	0.16	0.20	0.14
Spring-RCP	-0.09	-0.25*	-0.30**	-0.22**	0.07	0.05	0.20
SweetHome	-0.02	**-0.57***	**-0.57***	**-0.59***	-0.14	-0.14	-0.11
Vuze	-0.09	-0.23**	-0.26**	-0.19**	0.07	0.10	0.07
ALL	-0.13**	-0.17**	-0.20**	-0.13**	0.07*	0.08*	0.11**

Significant results obtained with $p - value < 0.01$ or $0.01 \leq p - value \leq 0.05$ are respectively annotated with ** or *. Significant correlations ($\rho \geq 0.5$ or $\rho \leq -0.5$, and $p - value \leq 0.05$) in bold face.

except in SweetHome. In this unique case, the negative correlation is strong enough (≈ 0.6). Finally, we observed a strong positive correlation between ISP and Cohesion (TCC/LCC) only in one application (Proguard). In both cases of strong correlations, the concerned applications have the smallest sets of interfaces, 13 for SweetHome and 18 for Proguard. As a consequence, and with regard to our findings F1 and F2 explained in the previous section, the low cohesion of software interfaces cannot be explained by conflicting forces with the other design properties of interfaces. These results lead us to the following finding:

F3 There is no evidence showing that the PTIP, ISP and Cohesion are conflicting properties of interface design, so that these properties can be achieved jointly. However, developers of interfaces abide by the PTIP and ISP properties, but neglect the Cohesion property.

This finding brings us to our final question, whether interfaces impact the cohesion of implementing classes.

Table V
SPEARMAN CORRELATION BETWEEN THE COHESION OF INTERFACES (i) AND THE COHESION OF THEIR IMPLEMENTING CLASSES, imp, AND DIRECT ONES, $dimp$ ($\alpha = 0.05$).

	Coh(i)*vs.*Coh(imp)			**Coh(i)*vs.*Coh($dimp$)**		
	TCC	LCC	SCOM	TCC	LCC	SCOM
ArgoUML	**0.60****	**0.61****	**0.56****	**0.60****	**0.61****	0.48**
Hibernate	0.44**	0.45**	0.40**	**0.55****	**0.56****	**0.58****
Jboss	**0.60****	**0.62****	**0.59****	**0.71****	**0.74****	**0.69****
Jfreechart	0.30**	0.35**	0.13	**0.51****	**0.52****	0.33**
JHotDraw	0.26**	0.24**	0.32**	**0.57****	**0.60****	**0.53****
Opentaps	0.14*	0.17**	0.22**	0.19*	0.22**	0.24**
PlantUML	0.44**	0.42**	0.37**	0.44**	0.40**	0.44**
Proguard	0.04	0.03	0.19**	0.02	0.01	0.17**
Rapidminer	0.17**	0.20**	0.14**	0.32**	0.38**	0.37**
Spring-RCP	0.36**	0.30**	0.38**	**0.58****	**0.52****	**0.57****
SweetHome	**0.63****	**0.75****	**0.52****	**0.68****	**0.76****	**0.57****
Vuze	0.46**	0.49**	0.47**	**0.52****	**0.55****	**0.52****
ALL	0.44**	0.45**	0.44**	**0.50****	**0.51****	**0.51****

Significant results obtained with $p - value < 0.01$ or $0.01 \leq p - value \leq 0.05$ are respectively annotated with ** or *. Significant correlations ($\rho \geq 0.5$ or $\rho \leq -0.5$ and $p - value \leq 0.05$) are in boldface.

C. Interface Design vs. Implementations Cohesion

Up to now, we found that interfaces are in general characterized by a low cohesion that is weakly correlated with the other design properties PTIP and ISP. In this section, we study whether this fact impacts on the cohesion of implementing classes, or not, by assessing the null hypothesis $H_{0Coh(int,imp)}$ presented in Section V-C.

To assess this hypothesis, we first study the correlation between the cohesion of interfaces and the cohesion of implementing classes. We perform this study with regard to all implementing classes, and for all 'Direct' implementing classes (see Section II). Then, we perform a Wilcoxon test comparing the cohesion of implementing classes to the cohesion of the other *concrete* (not-implementing) classes.

Note. Recall that an interface i can have several implementing classes. Thus, to study the correlation between interface cohesion and implementation cohesion, with high precision, we consider for each interface i all the pairs of cohesion values with regard to i's implementations: (Coh(i),Coh(imp_1)), (Coh(i),Coh(imp_2)) ... (Coh(i),Coh(imp_n)), where 'Coh' refers to a cohesion metric (TCC, LCC or SCOM), and where an imp_j denotes an implementing class to i. In other words, we do not simply use the average/median cohesion value of implementations to perform the correlation analysis.

The results in Table V show the following observations. The correlation between the cohesion of interfaces and the cohesion of all their implementing classes (first three columns) is in general statistically significant, but generally below 0.5. When considering all the applications, the correlation is average (≈ 0.45). However, by considering only direct implementing classes (last three columns), the correlation becomes stronger in almost all cases. This shows that there is considerable association between interfaces' design and the cohesion of their implementing classes, particularly those classes which directly implement interfaces. This is true despite the variability in the semantics and behaviors of those classes, their number and size, and their freedom to implement different interfaces at the same time and/or to belong to different class-type hierarchies (recall our discussion in Section IV-C).

From another perspective, Table VI shows the difference between cohesion mean values for implementing classes and not-implementing classes. At first glance, the delta values are not that big, and we cannot assume that using interfaces always impacts (degrades) the cohesion of classes. Actually, in many cases, the mean cohesion values of classes implementing interfaces is somewhat larger than the cohesion of those which do not implement interfaces. Notable exceptions are ArgoUML, Hibernate, JBoss, PlantUML, Rapidminer, and Vuze. In these cases, the results show that the cohesion of implementing classes is worse than the cohesion of the other classes. Surprisingly, these cases match exactly the cases in which interfaces are characterized with low cohesion: see Table III and our discussion in Section VI-A. This confirms that our design of interfaces impacts the cohesion of classes which implement them. This is also supported by the results of the Wilcoxon test comparing the cohesion of all implementing classes against that of all not-implementing classes. The results show a statistically significant degradation of the cohesion of implementing classes compared the cohesion of not-implementing ones. In addition to support our finding about interface cohesion F2, these results allows us to reject the null hypothesis $H_{0Coh(int,imp)}$ and accept the alternative one $H_{1Coh(int,imp)}$:

F4 Interfaces negatively impact the cohesion of classes.

However, considering that the deltas between cohesion mean values are large enough only for applications with low mean cohesion for interfaces, we claim that interfaces impact negatively the cohesion of implementing classes if interfaces do not adhere to the Cohesion property.

Results Summary

Our study shows that there is empirical evidence that software developers definitely abide by the "Program to Interface" and "Interface Segregation" principles of interface design, but they neglect the "Cohesion" property. However, there is no significant evidence to explain the low cohesion at interfaces by an intrinsic conflict with the well-known principles of interface design (aforementioned). This empirical finding is of high importance since the results show that there is considerable correlation between the cohesion of interfaces and the cohesion of their implementing classes, particularly, direct implementation ones. Note that this result was obtained while considering all varieties of interfaces, including those declaring just a couple of methods; and regardless of the size and number of implementing classes, or other factors that can cause a strong variability in the semantics and behaviors of (variability in the cohesion of) implementing classes. Finally, and most important, there is empirical evidence that such design practices of interfaces lead to a degraded cohesion of implementing classes, where the cohesion of these latter (as measured by the TCC, LCC and SCOM metrics) is worse than the cohesion of not-implementing classes.

Results Implications

We believe our findings are interesting for both researchers and software developers (and maintainers). Although the "Program to Interface" and "Interface Segregation" principles (PTIP and ISP) of interface design are widely known, this is, to the best of our knowledge, the first time in software engineering research that the results show strong evidence on

Table VI
COMPARISON BETWEEN THE COHESION OF IMPLEMENTING CLASSES (IMP: 5256 CLASSES IN TOTAL) AND THAT OF NOT-IMPLEMENTING CLASSES (NOT_IMP: 11161 CLASSES IN TOTAL), WITH TWO-TAILED WILCOXON TEST ($\alpha = 0.05$).

	$\overline{\text{TCC}}$			$\overline{\text{LCC}}$			$\overline{\text{SCOM}}$		
	Imp	**NOT_Imp**	Δ	**Imp**	**NOT_Imp**	Δ	**Imp**	**NOT_Imp**	Δ
ArgoUML	0.19	0.32	**-0.13**	0.21	0.34	**-0.13**	0.13	0.29	**-0.16**
Hibernate	0.36	0.46	**-0.10**	0.37	0.48	**-0.11**	0.30	0.45	**-0.15**
Jboss	0.30	0.34	**-0.04**	0.36	0.45	**-0.09**	0.20	0.27	**-0.07**
Jfreechart	0.55	0.48	*0.07*	0.56	0.47	*0.09*	0.47	0.46	*0.01*
JHotDraw	0.38	0.44	**-0.06**	0.42	0.35	*0.07*	0.23	0.37	**-0.14**
Opentaps	0.51	0.47	*0.04*	0.50	0.48	*0.02*	0.43	0.35	*0.08*
PlantUML	0.40	0.48	**-0.08**	0.40	0.45	**-0.05**	0.42	0.45	**-0.03**
Proguard	0.33	0.33	0.00	0.33	0.35	**-0.02**	0.29	0.29	0.00
Rapidminer	0.32	0.36	**-0.04**	0.32	0.35	**-0.03**	0.29	0.29	0.00
Spring-RCP	0.45	0.45	0.00	0.47	0.41	*0.06*	0.37	0.37	0.00
SweetHome	0.42	0.41	*0.01*	0.53	0.42	*0.11*	0.18	0.32	**-0.14**
Vuze	0.31	0.39	**-0.08**	0.23	0.36	**-0.13**	0.22	0.33	**-0.11**
ALL	0.34	0.41	W.diff. **-0.07****	0.36	0.41	W.diff. **-0.05****	0.27	0.35	W.diff. **-0.07****

Delta values in **boldface** (*italic-face*) denote that implementing classes have **worse** (*better*) cohesion than not-implementing classes. For ALL applications, the *W.diff.* denotes the *Difference-in-Location* obtained from a two-tailed Wilcoxon test. ** and * denote respectively results obtained with $p - value < 0.01$ or $0.01 \leq p - value \leq 0.05$.

the adherence of interfaces to these principles in real and large long-living software systems. Hence, this paper provides a strong foundation for the importance of these design principles for further investigation of interface design and OO program quality. The results imply that researchers and software maintainers should consider the particularities of interfaces by using appropriate metrics for assessing interfaces' design (e.g., SIUC metric and our proposed metric LPTI).

Furthermore, the results of our study imply that researchers and software maintainers should consider the impact of interface design on implementing classes while investigating their quality. They should consider the Cohesion property, in addition to other design principles, while designing and/or maintaining interfaces. That is by using appropriate models, such as our proposed model IIM, for mapping class metrics (e.g., cohesion and coupling metrics) for assessing program quality at the interface level. This would assist maintainers in identifying interfaces that cause a poor design quality. Additionally, our findings suggest that researchers in software refactroing field should consider the refactoring of interfaces (along with their implementing classes) to improve programs' design. Finally, due to the important cost that may be caused by changes in interfaces, our findings strongly suggest that researchers should investigate new methodologies (e.g., metrics) that can estimate the internal cohesion of interfaces in early design stages (i.e., before realizing implementing classes).

VII. THREATS TO VALIDITY

This section considers different threats to validity of our findings.

The threats to internal validity of our study concern the used variables, such as the values of the metrics (LPTI, SIUC, TCC, LCC and SCOM). In our study, the values of all used metrics are computed by a static analysis tool (Moose [29]) and deterministic functions. We chose Moose because its metamodel, which is FAMIX, includes descriptions of all software entities (e.g., interfaces, classes, methods, attributes, etc.) and their associated relationships that are necessary to compute our metrics. In the presence of programming-language dynamic features such as polymorphism and dynamic class loading, the dependencies (e.g., method calls) could be slightly over- or underestimated by static analysis [30]. In our work, this threat does not concern the computed values of LPTI since this metric is computed using only static type references Section IV-A. Regarding the SIUC values, this threat is mitigated since we well handled the problem of polymorphic methods calls, that is due to late-binding mechanism, as explained in Section II (see *"Interface Segregation Principle"* paragraph). As for the values of used cohesion metrics, this threat is very mitigated since computing these metrics is based only on local attribute accesses and local method calls –i.e., via `this/self` or `super`, see Table I.

Construct validity threats can come from the fact that the used metrics might not actually measure the studied properties. Regarding the class cohesion metrics, we circumvent these threats by carefully selecting various cohesion metrics. Our selections were based on relevant recent studies about class cohesion metrics [13], [12]. We derived our finding about the Cohesion property by considering all of these cohesion metrics. As for the SIUC metric, we referred to a recent published study using also this metric for assessing the alignment of interface design to the design principle ISP [3]. The only metric we defined in this paper is LPTI, since we did not find in literature any metric for assessing the alignment of interface design to the PTIP. Still, in our definition of LPTI, we carefully referred to the interpretations of the PTIP in most relevant studies about interface design principles (e.g., [5], [1], [2], [4]).

Threats to external validity concern the generalization of our findings. One possible threat in our study is that all the used applications are open-source software systems. We believe, however, that our sample is fairly representative because we carefully selected twelve applications that differ in size and utility, among them some systems are widely used in both academic and industry (e.g., Hibernate, Jboss and Rapidminer). We plan, in the near future, to verify our findings through additional industrial projects.

To prevent the possible threats to conclusion validity, we carefully select the statistical methods that are adequate for both the nature of the data and the hypotheses. Because our data is not normally distributed (Anderson-Darling test), we used the Spearman correlation coefficient to investigate the

relation between different interface properties (PITP, ISP and Cohesion) and also to investigate the relation between interface cohesion and the cohesion of implementing classes. We used the non-parametric statistical hypothesis test, Wilcoxon signed-rank test to compare the cohesion of implementing classes with that of other classes. Finally, we performed many tests, and obtained p-values were in general very low. After their correction for multi-tests, they remain statistically significant.

VIII. Conclusion and Future Work

In this paper, we investigated the design of software interfaces from the perspective of the well-known design principles "Program to an Interface, not an Implementation" (PTIP) and "Interface Segregation" (ISP), and with regard to the Cohesion property. We conducted an empirical study through a large variety of open-source software systems containing 2050 interfaces and many thousands of classes. For our study, we used the SIUC metric and defined the LPTI metric to assess the conformance of interfaces to the PTIP and ISP principles. Furthermore, we used several class cohesion metrics (TCC, LCC and SCOM) for assessing the cohesion of classes, and we adapted them for the interfaces by means of the Interface-Implementations Model (IIM), that we propose in this paper.

The results of our study showed that software developers abide well by the design principles PTIP and ISP, but they consider less the cohesion property. This empirical finding is of high importance since the results also showed that interfaces with low cohesion tend to degrade the cohesion of all classes implementing them, compared to the classes which do not implement interfaces. The results of our study imply that researchers and software maintainers should consider the impact of interface design on programs while investigating their quality. They should also consider the Cohesion property, in addition to other design principles, while designing and/or maintaining interfaces. Our findings suggest that existing monitoring tools of software quality should be extended to adapt the class cohesion metrics for assessing the cohesion of interfaces. This would assist maintainers in identifying interfaces that cause a poor design quality. In addition, tools can provide information about interface use by using the the LPTI metric for assessing the alignment of interfaces to the PTIP.

Our future work will concentrate on investigating further interface properties, such as size, depth of inheritance and number of implementing classes, and the relations between these properties and the properties that we investigated in this paper. Another direction to investigate in the future is the associations between interface design and external quality attributes, such as program reusability and comprehensibility.

Acknowledgment

This publication was made possible by NPRP grant #09-1205-2-470 from the Qatar National Research Fund (a member of Qatar Foundation).

References

[1] B. Venners, "Designing with interfaces," 1998, http://www.javaworld.com.

[2] R. C. Martin, "Design principles and design patterns," 2000, www.objectmentor.com.

[3] D. Romano and M. Pinzger, "Using source code metrics to predict change-prone java interfaces," in *Proceeding of ICSM'11*, 2011, pp. 303–312.

[4] N. Warren and P. Bishop, *Java in Practice*. Addison Wesley, 1999.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[6] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Trans. on Softw. Eng.*, vol. 20, no. 6, pp. 476–493, Jun. 1994.

[7] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.

[8] T. Mens and T. Tourwé, "A survey of software refactoring," *Trans. on Softw. Eng.*, vol. 30, no. 2, pp. 126–138, 2004.

[9] P. Mihancea and R. Marinescu, "Discovering comprehension pitfalls in class hierarchies," in *Proceedings of CSMR '09*, 2009, pp. 7–16.

[10] B. Liskov, "Data abstraction and hierarchy," in *Proceeding of the OOPSLA'87*, ser. OOPSLA '87, 1987, pp. 17–34.

[11] J. M. Bieman and B.-K. Kang, "Cohesion and reuse in an object-oriented system," in *Proceedings of the 1995 Symposium on Softw. reusability*, ser. SSR '95. ACM, 1995, pp. 259–262.

[12] L. H. Etzkorn, S. Gholston, J. Fortune, C. Stein, D. R. Utley, P. A. Farrington, and G. W. Cox, "A comparison of cohesion metrics for object-oriented systems," *Inf. & Softw. Tech.*, vol. 46, no. 10, pp. 677–687, 2004.

[13] J. Al Dallal, "Measuring the discriminative power of object-oriented class cohesion metrics," *IEEE Trans. Softw. Eng.*, vol. 37, no. 6, pp. 788–804, Nov. 2011.

[14] M. Perepletchikov, C. Ryan, and K. Frampton, "Cohesion metrics for predicting maintainability of service-oriented software," in *Proceedings of QSIC '07*, ser. QSIC '07. IEEE Computer Society, 2007, pp. 328–335.

[15] L. Fernández and R. P. na, "A sensitive metric of class cohesion," *Inf. Theories and Applications*, vol. 13, pp. 82–91, 2006.

[16] B. Henderson-Sellers, *Object-Oriented Metrics: Measures of Complexity*. Prentice-Hall, 1996.

[17] W. Li and S. Henry, "Maintenance metrics for the object oriented paradigm," in *Proceedings of International Softw. Metrics Symposium*, 1993, pp. 52–60.

[18] M. Hitz and B. Montazeri, "Measuring coupling and cohesion in object-oriented systems," in *Proc. Intl. Sym. on Applied Corporate Computing*, 1995.

[19] H. Liu, Z. Ma, W. Shao, and Z. Niu, "Schedule of bad smell detection and resolution: A new way to save effort," *IEEE Trans. on Softw. Eng.*, vol. 38, no. 1, pp. 220–235, Jan. 2012.

[20] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," *IEEE Trans. on Softw. Engineering*, vol. 38, pp. 5–18, 2012.

[21] A. Ouni, M. Kessentini, H. A. Sahraoui, and M. Boukadoum, "Maintainability defects detection and correction: a multi-objective approach," *Autom. Softw. Eng.*, vol. 20, no. 1, pp. 47–79, 2013.

[22] M. Kessentini, S. Vaucher, and H. A. Sahraoui, "Deviance from perfection is a better criterion than closeness to evil when identifying risky code," in *25th IEEE/ACM International Conference on Automated Software Engineering*, 2010, pp. 113–122.

[23] M. Abbes, F. Khomh, Y.-G. Gueheneuc, and G. Antoniol, "An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension," in *Proceedings of CSMR'11*, ser. CSMR '11. IEEE Computer Society, 2011, pp. 181–190.

[24] H. Abdeen and O. Shata, "Characterizing and evaluating the impact of software interface clones," *International Journal of Software Engineering & Applications (IJSEA)*, vol. 4, no. 1, pp. 67–77, Jan 2013.

[25] M. A. S. Boxall and S. Araban, "Interface metrics for reusability analysis of components," in *Proceedings of ASWEC '04*, ser. ASWEC '04. IEEE Computer Society, 2004, pp. 40–51.

[26] H. Abdeen and O. Shata, "Metrics for assessing the design of software interfaces," *International Journal of Advanced Research in Computer and Communication Engineering*, vol. 1, no. 10, pp. 737–745, dec 2012.

[27] D. Kawrykow and M. Robillard, "Improving api usage through automatic detection of redundant code," in *Proceedings of ASE'09*. IEEE, 2009, pp. 111–122.

[28] M. Cataldo, C. de Souza, D. Bentolila, T. Miranda, and S. Nambiar, "The impact of interface complexity on failures: an empirical analysis and implications for tool design," School of Computer Science, Carnegie Mellon University, Tech. Rep., 2010.

[29] S. Ducasse, M. Lanza, and S. Tichelaar, "Moose: an Extensible Language-Independent Environment for Reengineering Object-Oriented Systems," in *Proceedings of CoSET '00 (2nd International Symposium on Constructing Software Engineering Tools)*, Jun. 2000.

[30] S. Allier, S. Vaucher, B. Dufour, and H. A. Sahraoui, "Deriving coupling metrics from call graphs," in *10th Working Conference on Source Code Analysis and Manipulation*, 2010, pp. 43–52.