

A concern-oriented framework for dynamic measurements



Walter Cazzola^{a,*}, Alessandro Marchetto^b

^a Department of Computer Science, Università degli Studi di Milano, Italy

^b Fondazione Bruno Kessler, Trento, Italy

ARTICLE INFO

Article history:

Received 30 July 2013

Received in revised form 19 August 2014

Accepted 20 August 2014

Available online 28 August 2014

Keywords:

Software measurements and metrics

Static and dynamic software artifact analysis

Software feature and concern

ABSTRACT

Evolving software programs requires that software developers reason *quantitatively* about the modularity impact of several concerns, which are often scattered over the system. To this respect, concern-oriented software analysis is rising to a dominant position in software development. Hence, measurement techniques play a fundamental role in assessing the concern modularity of a software system. Unfortunately, existing measurements are still fundamentally module-oriented rather than concern-oriented. Moreover, the few available concern-oriented metrics are defined in a non-systematic and shared way and mainly focus on static properties of a concern, even if many properties can only be accurately quantified at run-time. Hence, novel concern-oriented measurements and, in particular, shared and systematic ways to define them are still welcome. This paper poses the basis for a unified framework for concern-driven measurement. The framework provides a basic terminology and criteria for defining novel concern metrics. To evaluate the framework feasibility and effectiveness, we have shown how it can be used to adapt some classic metrics to quantify concerns and in particular to instantiate new dynamic concern metrics from their static counterparts.

© 2014 Elsevier B.V. All rights reserved.

1. Introduction

A *concern* is any consideration that can affect the implementation and maintenance of program modules [1]. In particular, a *concern* is identified by portions of code not necessarily contiguous that contribute to implement such a concern; the concern can be selectively exercised through *ad hoc* scenarios defined by, e.g., use cases or test units. A software requirement or functionality, for instance, is a concern while the dynamic counterpart is the execution of a requirement or functionality. As an example, the services provided to the user by a software system that controls an automated teller machine (ATM) are concerns.

Normally, the software is developed reasoning in term of the features¹ it must provide but the tangled nature of the resulting application forces the maintainer to reason quantitatively about their modularity to facilitate its maintenance. With the increasing relevance of concern-oriented programming, see, for example, the advent of aspect-oriented programming (AOP) [2] and feature-oriented programming (FOP) [3], there is an urge to revise existing metrics (as done by [4]) and to develop new ones supporting concern

quantification against software variability. For instance, some studies [5,6] suggested that an increment to software modularity might correspond to: (i) an increment of undesirable couplings involving the realization of two or more concerns; and (ii) a decrement of the cohesion among the elements realizing a concern. *This kind of concern-specific design anomalies are key factors to decrease software maintainability.*

However, to provide an accurate characterization of how a concern affects a program is not a trivial task [4]. Many concerns are often tangled and scattered across a number of modules and, therefore, there is no direct traceability between a concern and the module boundaries [1]. The mapping between concern and code modules—i.e., “where the concern is implemented in the code”—is not always well-documented and well-preserved during the system design, implementation and maintenance. In such cases, the mapping between concerns and code modules can be inferred by static code analysis (to the static extent) and completed by dynamically exercising the concern, e.g., via test units (to the dynamic extent) [7]. As a result, concern-specific properties cannot be detected by applying conventional module-oriented metrics and proper variants of such metrics have been investigated in the literature, such as [4,8,9].

By analyzing the existing literature in the field of concern-oriented metrics, however, we observed two main limitations:

* Corresponding author.

E-mail addresses: cazzola@di.unimi.it (W. Cazzola), alex.marchetto@gmail.com (A. Marchetto).

¹ In the rest of the paper, *feature* and *concern* will be used as synonyms.

1. Existing metrics are not systematically defined, that is, there is a lack of shared frameworks or approaches that can support the systematic definition of concern-oriented metrics. In fact, to the best of our knowledge, there exists only one measurement framework (i.e., the one described in [9]) devoted to design and describe concern-oriented metrics; all the others frameworks available in the literature—such as [10,11]—only support module-oriented metrics, thus they cannot be reused as-is to define new concern-oriented metrics. Consequently, designers of measurement tools cannot rely on formal, systematic and shared terminology, set of notions and criteria to: define and describe concern metrics and systematically validate and compare them, e.g., with the existing ones. This leads to ambiguous and overlapping metric definition that hampers the adoption of concern metrics in academic and industry settings and the execution of empirical studies using these metrics in general.
2. Existing concern-oriented metrics are mainly static, i.e., they quantify statically-computable properties of a concern, as we have identified in a recent systematic study [9]. However, as happens in the case of software modules [10,11], some relevant properties of a concern can only be precisely discovered though the concern execution [4], such as dynamic coupling or cohesion. Static and dynamic metrics are hence complementary also at concern-level as well as at module-level. In fact, static metrics are conservative and can lose precision since they are based on static analysis of software artifacts (e.g., source code), while dynamic metrics are strongly tied to specific software executions, thus they can be more precise than the static ones but they can suffer of under-approximated results, i.e., the part of the system not executed is not considered in the metric computation.

This paper presents a contribution in this field by providing a concern-driven framework for defining and describing both static and dynamic metrics, at both module and concern levels. In particular, the presented framework extends and complements our measurement framework presented in [9] by capturing run-time properties that can be quantified for a concern and how they can be obtained. The framework is composed of a group of terms, notions and criteria for defining and comparing dynamic concern metrics beyond those for defining and comparing static concern metrics.

We evaluated the presented framework's feasibility and effectiveness in two ways. First, we conducted an experiment (Section 6) where some subjects (students) have used the framework to instantiate some dynamic concern-oriented metrics from their static or module-oriented counterparts; the goal of this experiment was to answer the research question: (RQ1): "*Can the framework be used to describe several concern-oriented metrics using a common and precise terminology and set of concepts?*". Second, we reported on a case study (Section 7) where we used some dynamic and static concern oriented metric to measure a pool of open source applications; the case study has been carried out with the goal of answering to the research question: (RQ2) "*Are the dynamic concern-oriented metrics useful to predict the concern bug-proneness?*". This case study aimed at showing utility and effectiveness of such dynamic concern metrics for bug-proneness estimation.

The rest of the paper is organized as follows. In Section 2 we present a survey of existing maintainability measurements and describe their adaptation to quantify dynamic properties. Furthermore, we stress the relevance of dynamic measurement by examples. In Section 3 we analyze the specific characteristics of measuring concerns dynamically, that are in particular, concern mapping and triggering, as well as a tool supporting the identification of a concern and its components at runtime. We introduce the

framework in Section 4 and the criteria composing it in Section 5. Section 6 provides an experimental evaluation of the proposed framework by metrics instantiation while Section 7 reports a study we conducted about the usage of dynamic measurements instantiated at concern-level through the presented framework. Finally, Section 8 summarizes the state-of-the-art about metric frameworks, and in Section 9 we draw our concluding remarks.

2. Towards dynamic concern measurement

To support dynamic concern-driven metrics definition and measurement we had to understand which properties and notions characterize a concern at run-time and whether it is worth measuring. Since the literature on dynamic concern measurement is scarce² we have looked at the literature about metrics (both at module and concern level) and dynamic properties of software systems for identifying such properties and characteristics. Therefore, to have a wide and comprehensive understanding of the concern's properties, we studied and adapted some existing static concern metrics and some well-accepted module-oriented metrics to quantify dynamic concern properties. Such an approach permitted to cover a larger amount of possible measurements and relevant properties that might otherwise be overlooked. Out of these findings, then, we defined a set of framework criteria that capture such properties and that make the framework complete and effective enough to describe existing and new dynamic concern-oriented metrics.

In the rest of this section we present the result of our investigation, in particular we show how the considered metrics have been adapted to the dynamic and/or concern-oriented context. We have classified the considered metrics according to their original characteristics as follows:

- Dynamic module metrics. These are dynamic module-driven metrics originally defined for object-oriented systems. They were adapted or extended to be applied to concerns as well.
- Static concern metrics. These are static metrics originally defined for concerns. They were adapted or extended to be dynamically applied.
- Dynamic concern metrics. These are dynamic metrics already defined for concerns that do not require any adaptation.

Table 1 summarizes the result of the literature review we conducted. The table shows the considered suite of metrics and it reports for each metric the original definition (column "Original Definition") present in the literature and the definition obtained from our adaptation (column "Modified Definition"). To complete the picture, in Table 2 we report the definition of those metrics that are already defined as dynamic and concern-oriented and therefore that do not need any adaptation in order to be considered.

The adaptation process is quite straightforward and relies on the adoption of the concept of *concern execution* that corresponds to the execution of the elements composing the concern that can be prodded by, for example, an *ad hoc* use case or test unit. If the considered metric is dynamic but not concern-oriented we mapped the subject and/or the target of the measurement to the concerns; whereas if the metric is already concern-oriented but not dynamic we have exclusively considered the events that occur during the execution. For instance, in *Concern Diffusion over Operations (CDO)* we look for components that participate in the concern

² To the best of our knowledge, [4] is the most relevant piece of work in this field by introducing *disparity*, *concentration* and *dedication* metrics.

Table 1
Metric suite summary: original (2nd col.) vs. modified (3rd col.) definition.

Dynamic module metrics		
Metric	Original definition	Modified definition
dCBOoC	dynamic Coupling Between Objects (dCBO) [12] counts, for a class, the number of couples with other classes at run-time	dynamic Coupling Between Objects over Concerns (dCBOoC) counts, for a concern, the number of couples between an element ^a with other elements at run-time. Note that the considered elements can be tied to different concerns
dLCoMoC	dynamic simple Lack of Cohesion in Methods (dLCoM) [12] for a class is the number of pairs of methods in the class that have no instance variables in common minus the number of pairs of methods that have common instance variables at run-time. When this value is negative, the metric value is set to 0	dynamic simple Lack of Cohesion in Methods over Concerns (dLCoMoC) for a concern is the number of pairs of methods in the concern that have no instance variables in common minus the number of pairs of methods that have common instance variables at run-time. When this value is negative, the metric value is set to 0
LCoC	Live Code (LC) [13] measures the number of bytecode instructions that are executed	Live Code over Concerns (LCoC) measures the total bytecode instructions exercised during a concern execution
NToC	Number of threads (NT) [13] counts the largest number of threads simultaneously active or running	Number of threads over Concerns (NToC) counts the largest number of threads simultaneously active or running during the execution of a given concern
Static concern metrics (aspect, feature, and property terms are used as synonym of concern)		
Metric	Original definition	Modified definition
dCD	Crosscutting Degree of a Concern (CD) counts the number of components affected by the pointcuts and by the introductions in a given aspect	dynamic Crosscutting Degree of a Concern (dCD) counts the number of executed components that are affected by the pointcuts and by the introductions in a given concern
dCDC	Concern Diffusion over Components (CDC) [14] counts the number of primary components that mainly contribute to the implementation of a concern and the number of components that access the primary components by using them in attribute declarations, formal parameters, return types, throws declarations and local variables, or call their methods	dynamic Concern Diffusion over Components (dCDC) counts the number of primary components that contribute to the implementation of a concern that are exercised during the concern execution and the number of components that access the primary components by using them in attribute declarations, formal parameters, return types, throws declarations and local variables, or call their methods during the concern execution
dCDO	Concern Diffusion over Operations (CDO) [14] counts the number of operations whose main purpose is to contribute to the implementation of a concern. In addition, it counts the number of methods, constructors, and advice that access any primary component of the concern by accessing their attributes, calling their operations or using them in parameters, return types, declarations and statements	dynamic Concern Diffusion over Operations (dCDO) counts the number of operations whose main purpose is to contribute to the implementation of a concern exercised during the concern execution. In addition, it counts the number of methods, constructors, and advice that access any primary component of the concern by accessing their attributes or calling their operations during the concern execution
dCSC	Concern Sensitive Coupling (CSC) [9] quantifies the number of server components that a concern realized by a given client component is coupled to. In other words, CSC counts the number of explicit connections that are associated to the concern in a component	dynamic Concern Sensitive Coupling (dCSC) quantifies the number of server components that a concern realized by a given client component is coupled to during the concern execution. In other words, dCSC counts the number of explicit connections in the system execution that are associated to the concern in a component
dFCD	Feature Crosscutting Degree (FCD) [15] counts the number of classes that are crosscut by a feature	dynamic Feature Crosscutting Degree (dFCD) counts the number of class components exercised at run-time that are crosscut by more than one concern
dLCC	Lack of Concern-based Cohesion (LCC) [9] counts the number of concerns addressed by the assessed component	dynamic Lack of Concern-based Cohesion (dLCC) counts the number of concern addressed by the assessed component during the execution
dSize	Size [16] counts the number of methods and attributes of a class associated to a property	dynamic Size (dSize) counts the number of operations and attributes of a component associated to a concern during its execution
dSpread	Spread [16] counts the number of classes related/tied to a given property	dynamic Spread (dSpread) it counts the number of the components related/tied to a concern and exercised at run-time

^a A concern component is one of the elements (classes, methods, ...) composing the concern. Since we are interested to the concern execution we consider their dynamic counterpart (objects, method calls, field accesses, ...).

Table 2
Metric suite summary (cont'd): Wong's [4] metrics.

Metric	Definition
Given	
	<ul style="list-style-type: none"> B_f is the set of execution slices of P (program) used to implement the feature f, and B_c is the set of execution slices in c component of P where an execution slice is a portion of the program code executed by an input that exercises a feature
Disparity	The disparity measures how close a feature f is to a program component c . It holds 1 when $B_f \cap B_c = \emptyset$. i.e., if and only if none of the blocks that implement c are used to implement f ; it holds 0 when $B_f = B_c$, i.e., if and only if the component c implements only and totally the feature f
Concentration	The concentration measures how much a feature f is concentrated in a program component c . It holds 1 when $B_f \subseteq B_c$, i.e., if all the blocks used to implement f are in c ; whereas it holds 0 when $B_f \cap B_c = \emptyset$, i.e., if and only if none of the blocks that implement c are used to implement f
Dedication	The dedication measures how much a program component c is dedicated to a feature f . It holds 1 when $B_c \subseteq B_f$ i.e., if all the blocks used to implement c are also used to implement f ; whereas it holds 0 when $B_f \cap B_c = \emptyset$, i.e., if and only if none of the blocks that implement c are used to implement f

definition whereas in the *Dynamic Concern Diffusion over Operations* (dCDO) we narrow the metric definition to the components that participate in the concern definition and are exercised during the concern execution. Some further adjustments were necessary to

target the metric to the corresponding dynamic element, e.g., methods become method invocations.

The set of metrics in Table 1 has been analyzed to understand its main characterizing factors, such as the measured run-time

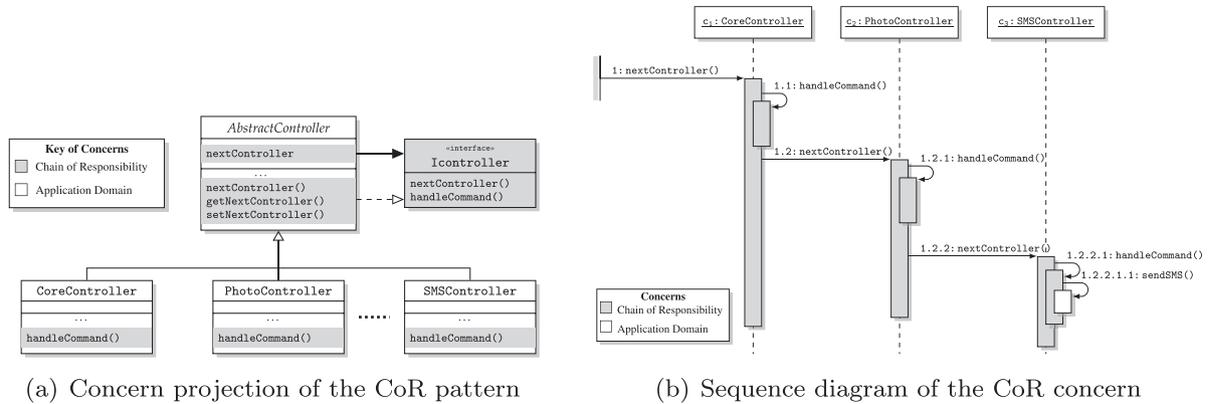


Fig. 1. The chain of responsibility (CoR) example.

properties and the counted software objects/components. These factors have to be included in our framework since they are necessary to describe a wide variety of dynamic concern-oriented metrics. Conversely, we do not validate these metrics any more since they are a straightforward variations of well-known metrics and their validation can be assessed from the original validation.

Summarizing, to capture and describe dynamic concern metrics, the framework must master all the basic elements necessary to deal with such kind of metrics. For example, the framework should provide the elements to answer this basic questions: how does a concern look at runtime? What are the basic elements composing a concern at runtime? How are such elements related/connected to each other? To answer such questions we analyzed the metrics listed in Table 1 to identify relevant concepts and notions characterizing dynamic aspects and properties of concern at runtime. For example, a software application can be seen as a set of components (e.g., classes or aspects) capable of interacting with each other. At runtime such components are replaced by their dynamic counterparts (i.e., instances of classes and aspects) and the potential connections are realized by message exchanges (i.e., method calls, effects at the join points, ...). Hence, we enrich our (initially static) framework [9] by means of this notion of “instance” of components. Then, we analyzed again the metrics listed in Table 1 to identify relevant aspects and properties of dynamic concerns (e.g., run-time properties and the counted software objects/components). Such aspects have been used, on one side, to extend and complement those aspects and properties already captured by the criteria of our framework (e.g., “instances” is a significant element in the dynamic metrics but it is not for static metrics; static and dynamic coupling [10] are distinctive relationships that exist among software elements that can be measured both statically and dynamically but with different results). On the other side, the identified aspects and properties have been used to look for (sub)criteria that our initial framework lacks to capture and/or to describe (e.g., concern projection into code [9] is inadequate to dynamically map system elements to concerns at runtime, in fact, due to code inheritance and polymorphism the effectively executed code can be substantially different from the one statically identified).

3. Concern mapping and triggering

Apart from the key factors necessary to describe them, concerns and their dynamic measurement introduce also two more aspects: (i) how to track code elements down in the target concerns and (ii) how to trigger such elements to exercise the concern in their definition. *The relevance of dynamic concern metrics.* Let us try to explain this on the object-oriented design of a product line for mobile device

Table 3
Static and dynamic measurements.

	CSC	dCSC
IController	0	0
AbstractController	1	0
CoreController	0	1
PhotoController	0	1
SMSController	0	0
Total	1	2

applications described in [17] and partially reported in Fig. 1. Fig. 1(a) shows a partial class diagram realizing the *Chain of Responsibility* (CoR) design pattern implemented in the product line. Inheritance relationships are extensively exploited by classes to make them play the pattern roles. For instance, `CoreController`, `PhotoController` and `SMSController` extend `AbstractController` which implements the `IController` interface.

Let us briefly analyze the case of the metrics *Concern Sensitive Coupling* (CSC) [14], as well as its dynamic counterpart dCSC (definitions are in Table 1). To compute CSC metric it is necessary: (i) to map the target concern into the code elements, (ii) to parse the identified code elements to extract the information required to compute the metrics (that is, the number of explicit connections associated to a concern in each element). Several techniques [7] exist to statically map concerns into code elements. All these methods are semi-automatic; they perform some (partial) source code analysis complemented by user intervention to identify those code elements related to a concern. Code elements realizing the CoR concern are shadowed in the design of Fig. 1(a) to quantify static concern metrics, such as CSC.

Table 3 presents the measurement of CSC for the excerpt of the design illustrated in Fig. 1(a). Due to the inheritance relationship, dynamic coupling, and polymorphism, the *effective* class of the object sending or receiving a message may be different from the class implementing the corresponding method. To make this clear, Fig. 1(b) shows a partial sequence diagram representing one possible execution scenario of this application. An execution scenario s is a sequence of interactions among system components/objects stimulated by input data or events and that realizes a system behavior, feature or functionality. The scenario in Fig. 1(b) represents a call chain from controller objects to the appropriate controller (in this case, c_3 which is an instance of the `SMSController` class) that will handle the request. Observing an execution of the CoR concern according to this scenario, or its sequence diagram (Fig. 1(b)), one can see that in addition to the static concern coupling between `AbstractController` and `IController` pointed out in the class diagram of Fig. 1(a), a

dynamic concern coupling exists (i.e., messages passing between controller objects couple their respective classes). For example, the CoR implementation (Fig. 1(b)) shows a message exchange between instances of `CoreController` and `PhotoController` that reveals that these classes are coupled. This information cannot be captured by a static code analysis, i.e., it cannot be considered for measuring CSC. Instead, it can be pointed out and captured by observing some executions of the target concern, i.e., it can be considered for measuring the dCSC. Since CSC and dCSC (Table 3) identify the static and dynamic concern couplings respectively they are obviously complementary. The CSC value for `AbstractController` is 1 since the code of the CoR concern in this class is—in the static view—coupled to `IController` interface. This connection is easy to spot in static diagrams, such as a UML class diagram (Fig. 1(a)). On the other hand, only a system execution is able to point out the connections identified by the dCSC metric, such as the dynamic concern coupling in `CoreController` and `PhotoController`.

3.1. How to measure dynamic concern metrics

It should be fairly evident that the dynamic measurement of a system can only be achieved by analyzing the running system. To this regard, a running object-oriented system is a set of class instances (*objects*) exchanging messages (*method invocations* or *field accesses*) to collaborate in realizing their tasks. The classic measurement techniques can apply also to concern-oriented (e.g., aspect- and feature-oriented) programs but different characteristics must be addressed, e.g., the behavior of a concern, how the concern is instantiated, and the effective coupling between concern and base code. A dynamic concern (or a concern at run-time) is, hence, strictly tied to the main concept of the dynamic analysis: the concern execution. In particular, it is identified by portions of code that are executed to realize a given concern (e.g., a given aspect, trait, feature, or functionality) and generally, at run-time a concern can be exercised by: (i) an application run; (ii) a trace, i.e., a part of the application run; (iii) a given scenario that can include many traces or (iv) a given set of scenarios. More formally, the dynamic measurement of a given concern involves the following main steps:

1. the definition of the execution scenario/s and inputs necessary to exercise the concern, e.g., a set of predefined test cases ([18,19] presented some approaches to automatically derive such scenarios);
2. the tracing of the concern execution and the recording of the relevant information via system instrumentation—as in [20,21]—or via some *ad hoc* run-time support—as by exploiting the JVMTI³;
3. the execution of the system to exercise the selected scenario.

During the concern execution, the mechanism installed in step 2 records a set of traces containing data about the system behavior. Off-line measurements are based on trace analysis after the system execution has been completed (e.g., see the metrics to measure the dynamic coupling [10,11]) whereas on-line measurements are done on-the-fly during or instead of the trace recording (e.g., see the metrics to measure the performance of a system, e.g., CPU load [10,11]).

Moreover, the dynamic measurement can be based on the execution of several execution scenarios and on several sets of inputs contemporaneously. To have a complete picture of the system con-

cern execution, we have to exercise as many usage scenarios as possible and in every possible execution context. Unfortunately, this is not always possible since such a combination of usage scenarios and contexts can be huge if not infinite: some coverage criteria, as shown by [22], can be successfully applied that will lead to a partial view of the concern behavior. Therefore, static and dynamic measurements have a different view of the system and also the measurements will differ accordingly. *Ad hoc* scenarios allow to exercise all and only the elements composing a given concern.

3.2. Tool support

In [20], Cazzola and Marchetto presented AOP \rightarrow HiddenMetrics: an Eclipse-based tool that supports the measurement of dynamic metrics for Java and AspectJ applications in a *noninvasive* way thanks to the use of aspect-oriented programming. Aspect-oriented programming provides a composition mechanism that permits to clearly separate the measurement process from the subject of the measurement avoiding code pollution or replication typical of traditional and more invasive approaches and, thus, widening the applicability of the measurement process. In AOP \rightarrow HiddenMetrics, the metrics are implemented as aspects and woven into the target application only when they have to be measured. Indeed, AOP \rightarrow HiddenMetrics uses a transparent plug/unplug mechanism to instrument the code of the target application with the measurement code (i.e., the aspects implementing the metrics). On one side, the tool supports different measurements (e.g., coupling on method calls, lack of cohesion of operations, code execution coverage) on several software properties (e.g., size, coupling, cohesion, memory use, code coverage) by means of predefined aspects. On the other side, it provides also an easy way to extend its set of metrics: to define new metrics requires only a bit of knowledge of aspect-oriented programming. A complete overview of AOP \rightarrow HiddenMetrics can be read in [20].

In this work, the AOP \rightarrow HiddenMetrics metrics set has been enriched with the concern-level metrics presented in Table 1 and the process needed to compute and measure them has been tuned. At this point, to measure an application the user has to:

1. define a set of system executions and code a set of test cases (e.g., in `jUnit`) able to exercise them;
2. choose the metrics to measure and weave the corresponding aspect/s to the target system; and
3. execute the test cases and wait for the aspect to collect the resulting measures.

The test case definition (i.e., step 1) is in charge of the user since it depends on the target system to measure; the remaining steps (i.e., step 2 and 3) are completely automated by AOP \rightarrow HiddenMetrics. See Section 7 for detailed examples of the tool usage.

4. Framework: basic concepts

This section presents the basic concepts used by our concern-oriented framework for dynamic measurements (Table 4 summarizes them). It also introduces a standard terminology which allows to express all the dynamic metrics in a consistent and meaningful manner, independently of the implementing language of the target system.

4.1. Concern and system elements

We introduce concepts and notions used to define a concern and the existing relationships between a concern and the structural elements of a system that realize such a concern. Moreover,

³ <http://docs.oracle.com/javase/6/docs/technotes/guides/jvmti>.

Table 4
Concern and system elements: summary of relevant notions.

Notation	Description
S	target system
$C(S), O(S)$	set of components and instances of S respectively
$M(c) = Att(c) \cup Op(c) \cup Dec(c)$	set of members (attributes, operations and declarations) of a component c
$M(ob) = Att(ob) \cup Op(ob)$	set of members (attributes and operations) of an instance ob
$Rt(op), Args(op), IP(op), St(op)$	set of return types, parameters, code-injection points and statements for an operation op
$Con(S)$	set of concerns of S
$M(con) = Att(con) \cup Op(con) \cup Dec(con)$	set of members (attributes, operations and declarations) of a concern con
$CC(S)$	set of the connections among components and instances of S
$CC(c) = EC(c) \cup IC(c)$	set of (explicit and implicit) connections of a component c with other components
$CC(o) = EC(o) \cup IC(o)$	set of (explicit and implicit) connections of an instance o with other instances
$Ancestors(c), Parents(c)$	sets of ancestors, parents, children and descendants of a component c
$Children(c), Descendants(c)$	
$AC(S), LC(S)$	partition of $C(S)$ and $O(S)$ among applications, and libraries

these concepts are helpful in specifying the key abstractions for a system, each type of abstraction is alternatively called an *element*.

A concern can be realized by an arbitrary set of elements of a system S . S consists of a set of components, denoted by $C(S)$. A component $c \in C(S)$ can be, for example, a class, an interface, an aspect, a feature or a set of these elements. At run-time, the counterpart of $C(S)$ is represented by a set of instances, i.e., objects denoted by $O(S)$, any $c \in C(S)$ can be instantiated by one or more objects $ob \in O(S)$ independently of its actual type (e.g., class). Aspects, features, traits and so on affect the instantiation process by enriching the final instance with data from a different concern. In general, the generic term *instance* is used to identify all kinds of objects when a more precise terminology is not necessary. The mapping $\phi : \mathcal{P}(O(S)) \rightarrow \mathcal{P}(C(S))$ connects an instance or a group to the component they are instantiated from. For example, if ob is an instance of the class c woven by the aspect a , $\phi(\{ob\})$ returns the set $\{c, a\}$. During the execution, an instance can be considered in isolation or aggregated at different granularity levels such as class-, scenario-, use-case-, and system-level. For example, Fig. 1(a) shows that the CoR pattern is statically composed of the abstract class `AbstractController`, the interface `IController`, and the classes: `CoreController`, `PhotoController`, `SMSController`. Instead, dynamically the CoR pattern, see the scenario in Fig. 1(b), is only composed of instances of such classes, e.g., interfaces cannot be instantiated into objects.

Each instance ob exposes a set of attributes, $Att(ob)$, and a set of operations, $Op(ob)$. From the dynamic perspective, hence, given $c \in C(S)$, the sets $Att(c)$ and $Op(c)$ are defined as the set of attributes and operations defined or provided by c . Note that operations are all those logical elements providing utilities, services and functionality. For instance are operations: the methods as defined in the object-oriented languages like Java; pointcuts and advices as defined in aspect-oriented languages⁴ like AspectJ; but also the mixins as used in feature-oriented languages (like Jak/AHEAD [23]) provide methods and method refinements that are considered as operations. Moreover, in the presence of dynamic weaving (as in CaesarJ [24]) and for dynamic object-oriented languages (e.g., Python) attributes and operations can be added to the objects at run-time and not only to the class; in such cases $Att(ob)$ and $Op(ob)$ can differ from the corresponding sets ($Att(c)$ and $Op(c)$) for the component c , ob is instance of c . The set $M(c)$ of members of c is defined by $M(c) = Att(c) \cup Op(c) \cup Dec(c)$, where $Dec(c)$ represents a set of declarations of the component c . Analogously, the set of the members of

an instance $M(ob)$ is defined as $M(ob) = Att(c) \cup Op(c)$. An operation $o \in Op(ob)$ can have a return type, $Rt(o)$, a set of parameters, $Args(o)$, a set of code-injection points IP (e.g., join points in aspect-oriented programming, mixins in feature-oriented programming), and a set of statements or executable lines of code, $St(o)$. At run-time, an operation o can be invoked and/or executed by a concern, an attribute a can be instantiated and/or accessed during the concern execution, a join point jp can be hit and thereby the woven advice executed, and mixins mxs are composed of feature-based software units by code synthesis. At run-time, a system can be viewed as a set of instances collaborating through messages to realize the system functionality. These collaborations are based on the connections $CC(S)$ existing between elements of the systems like instances or components. Hence, $CC(S)$ is composed of operation calls, attributes uses, effects at join points and so on.

Notice that software concerns are compositions of generic elements of a system. A concern is an abstraction addressed by those elements that have the purpose of realizing it. An example of concern is a software requirement or functionality while the dynamic counterpart of a concern is a requirement or a functionality that can be executed through, at least, a scenario. To have a concern-based measurement, it is necessary to associate each structural element of the system (e.g., components) to the concerns it is realizing. For instance, Fig. 1(a) shows the projection of the CoR pattern on the code (static point of view of the concern) and Fig. 1(b) shows one execution scenario of the CoR pattern that can be executed to map the concern to the elements realizing it (dynamic point of view of the concern), see Section 3.

The set of concerns addressed/implemented by the system S is defined as $Con(S)$. $SCN(con)$ is a set of scenarios which exercise the concern $con \in Con(S)$. $C(con)$ is the set of components realizing a concern. Instead, $Att(con)$ and $Op(con)$, are respectively the set of attributes and operations realizing a concern and so, as well as for components, the set of members of a concern con is defined as $M(con) = Att(con) \cup Op(con)$. To statically identify the set of members that implement a concern also a set of declarations, $Dec(con)$, must be considered: $M(con) = Att(con) \cup Op(con) \cup Dec(con)$.

4.2. Components and connections

The *connection* is a dependency relationship where an element (*server*) provides a service to another element (*client*). Two elements are dynamically connected when one of them sends a message to the other. Method and constructor invocations, attribute accesses and effects at the join points are examples of exchanged messages. This variety characterizes the connections among the elements.

⁴ As [2] defined them, pointcuts are language elements that capture the so-called join points,—i.e., well-defined points in the program flow such as method calls, object instantiations, and variable accesses—, where the advices are woven.

Table 5
Concern-oriented model instantiation.

Element	Static		
	Java	AspectJ	Jak
System	<i>system</i>	<i>system</i>	<i>system, system extension</i>
Concern	<i>concern</i>	<i>concern</i>	<i>concern</i>
Component	<i>classes and interfaces</i>	<i>classes, aspects and interfaces</i>	<i>classes, mixins, and their refinements and extensions</i>
Interface	<i>method signature</i>	<i>method signature</i>	<i>method signature, class/mixin specification</i>
Attribute	<i>fields and variables</i>	<i>fields, variables and inter-types</i>	<i>classes, mixins, fields and variables</i>
Operation	<i>methods and constructors</i>	<i>methods, constructors, inter-types and advices</i>	<i>methods, constructors and expressions</i>
Statement	<i>java instructions</i>	<i>java and aspectj instructions</i>	<i>jak instructions (like java inst.), expressions</i>
	Dynamic		
	Java	AspectJ	Jak
System	<i>scenarios, features</i>	<i>scenarios, features</i>	<i>composition of features</i>
Concern	<i>scenarios</i>	<i>scenarios</i>	<i>features</i>
Component	<i>class instances</i>	<i>class and aspect instances</i>	<i>class and mixin instances</i>
Interface	<i>reflective invocations*</i>	<i>reflective invocations*</i>	<i>reflective invocations* of component and their compositions</i>
Attribute	<i>usage of fields and variables</i>	<i>usage of fields, variables and inter-types</i>	<i>usage of classes, mixins, fields and variables</i>
Operation	<i>method and constructor invocations</i>	<i>methods, constructors, inter-types and advice invocations and advice executions</i>	<i>method, class, mixin and constructor invocations, expression executions</i>
Statement	<i>bytecode instructions**</i>	<i>bytecode instructions**</i>	<i>bytecode instructions**</i>

Two kinds of connections can be identified: *explicit*, and *implicit*. For instance, an *explicit* connection of a component c , denoted by $EC(c)$, is caused by elements of c calling an operation or accessing an attribute of another component. On the other hand, an *implicit* connection of a component c , denoted by $IC(c)$, is caused by hitting a woven join point during the execution or by a handler catching an exception. The set of connections of a component c is defined as $CC(c) = EC(c) \cup IC(c)$. Similarly, $CC(o) = EC(o) \cup IC(o)$ the set of connections of an instance o . A reflective invocation, such as `mtd.invoke(obj, args)`, introduces an implicit connection among, at least, three components: the class of the `obj` variable, that represents the *target* of the message, the class of the `this` element, that represents the *source* of the message and the class defining the method reified by the `mtd` variable, that represents the *context*. Note that the context and the target can differ at run-time due to, for example, inheritance and inter-type declarations. The objects involved as arguments are similarly coupled. Reflection hampers static code analysis since the involved components can be reflectively created from external inputs. Similar issues can be raised for many other language constructs, as AspectJ's `flow` and `if`, CaesarJ's `deploy` and Scala's mixed trait/class based inheritance mechanism. Furthermore, components may participate in inheritance relationships. Inheritance and late binding can lead to different measurements with respect to the static approach (as described in the examples of Section 3). To this respect, for a component c , the following sets are defined: (i) *Ancestors(c)*—all recursively defined parents; (ii) *Parents(c)*—the directly declared parents; (iii) *Children(c)*—the directly derived children, and (iv) *Descendants(c)*—the recursively derived children.

The set of all components of a system $C(S)$ can be seen as partitioned into two subsets according to the extent of each component in the system itself: components defined in the system, or in a library or framework belong to the application, that are respectively $AC(S)$ and $LC(S)$. Given one of these partitions it is also possible to determine the set of instances, attributes, and operations belonging to the partition, e.g., given $AC(S)$ its sets of instances, attributes and operations are $O(AC(S))$, $Att(AC(S))$ and $Op(AC(S))$, respectively. In other terms, $O(AC(S))$ represents the set of instances belonging to components part of the application. The mapping $\phi(O(AC(S)) : \mathcal{P}(O(S)) \rightarrow \mathcal{P}(AC(S) \cup LC(S))$ connects an instance or a group of instances to the part (i.e., $AC(S)$ or $LC(S)$) they are instantiated from. For example, if ob is an instance of the class c belonging to the application, i.e., $AC(S)$, and woven with an aspect a

belonging to a library, i.e., $LC(S)$, $\phi(ob)$ returns $AC(S)$ and $LC(S)$. Conversely, in case both c and a belonging to the application, i.e., $AC(S)$, $\phi(ob)$ returns only $AC(S)$.

4.3. Language mapping

The aforementioned concern-oriented model is abstract enough to be instantiated for different modeling and programming languages. Table 5 provides a brief example on how our model can be instantiated to Java, AspectJ and Jak programming languages. Moreover, it shows the differences between static (given in [9]) and dynamic instantiation on these languages. Most of the mappings are intuitive and derive directly from the framework description. Few notes can be added: * interfaces are typical concepts without a dynamic counterpart but reflective calls can be identified through them also during the application; ** even if the bytecode instructions can be easily reverted to their source counterparts, they have a different granularity (many bytecode instructions are out of a single Java instruction) and metrics such as LOC clearly have a different measurement unit; *** scenarios can be realized by test cases written in JUnit [25] or similar tools.

5. Framework: the criteria

This section presents the concern-oriented framework for dynamic measurement which relies on the notions introduced in the previous sections.

The framework is defined according to a set of criteria and sub-criteria we derived from our investigation on (dynamic) concern-oriented metrics, i.e., the one documented in Section 2 and Section 3. Each criterion comes with a small description explaining which is the criterion, its instantiation details and some examples showing how to use it to describe new metrics. This version of the framework is based on the framework presented in [9]. New criteria or sub-criteria needed for specifying dynamic measurements have been introduced when appropriate.

5.1. Entities of concern measurement

The entity of measurement determines the application elements that we are going to measure and for which we are measuring/evaluating a given property of interest. Dynamic measurement at the concern level is aimed to capture run-time characteristics or

properties of given concerns and to manipulate them in a formal way. This criterion, hence, defines the level at which the collected measurement information can be interpreted. For instance, the run-time “size” of a concern can be different from the conventional static size since the system elements actually executed can be different, or they can have different characteristics from those defined in the system and composing the target concern. The run-time size can be measured, e.g., in terms of concern instances or operations really exercised during a software execution and it gives us an idea about the size of a given system aspect at concern-level.

Criterion instantiation. Usually concern measurement adopts concerns as the entity of measurement, but other selections are also possible. Although all elements in the concern-oriented model (see Section 4) may be selected in this criterion, the most common entities of concern measurement are: (i) system, (ii) concern, (iii) component, (iv) instance, (v) attribute, and (vi) operation.

Example. If we are interested in knowing how much a given concern is spread over the operations of the application to measure, we have to measure how many operations are executed to realize such a concern. In this case, the entities of measurement is the “concern” since we are measuring a property of the concern of interest. Conversely, if we want to know how many operations are exercised if a component (e.g., class, feature or aspect) is instantiated the entity of measurement is “component”.

5.2. Concern-aware attributes

Attributes are the properties that a concern (or, more generally, an entity of measurement) possesses and in which we are interested in. If we observe the behavior of two concerns by using scenarios we can say, for example, that one is more spread than the other in the system. A concern metric allows us to capture the “*is more spread than*” relationship and map it to a formal system, enabling us to mathematically explore the relationship. An entity possesses many attributes and an attribute can qualify many different entities [15]. For example, size can apply to several different software entities, such as components, computational units, operations, or concerns. An additional factor, “*at run-time?*”—with “*yes*” or “*no*” as possible values—specifies if the properties should be observed at run-time.

Criterion instantiation. In the attribute selection we may choose any property of the entity that we want to measure. Possible values related to static properties (i.e., properties that can be observed without the system execution) can be: (i) scattering, (ii) tangling, (iii) closeness [4], (iv) coupling, (v) cohesion and (vi) size. Similarly, possible values related to dynamic properties of the system under analysis are: (i) run-time scattering,⁵ (ii) run-time tangling,⁶ (iii) run-time closeness [4], (iv) run-time coupling [10,12], (v) run-time cohesion [12], (vi) run-time size [13]; (vii) used memory [13]; (viii) and concurrency [13].

Example. If we are interested in knowing how much a given concern is spread over the operations of the application to measure, we are interested in measuring the “*scattering*” property of the concern. Conversely, if we want to know how many system elements are composing the concern, we are interested in measuring the “*size*” property of the target concern.

5.3. Units

A concern measurement unit determines how to measure an attribute. An attribute can be measured by one or more units and

the same unit may be used to measure more than one attribute. For example, the size of a concern at run-time might be measured by counting either the number of executed bytecode instructions, code statements, the number of components used during the execution of the scenario realizing the concern.

Criterion instantiation. Possible values are any quantifiable element as computational unit. For example, (i) concerns, (ii) components, (iii) operations, (iv) attributes, (v) lines of code, (vi) bytecode instructions, (vii) executable lines of code or statements, (viii) allocated memory, (ix) exchanged messages, (x) frequencies of statement execution, (xi) number of active threads, (xii) number of reached joint points, (xiii) advice executions, (xiv) execution time, (xv) instances.

Example. If we are interested in knowing how much a given concern is spread over the operations of the application to measure, we have to measure how many operations are used to realize such a concern. In this case, the entities of measurement is the “*operation*”. Instead, if we want to know how much the concern is spread over the components of the system, we have to work at “*component*” (e.g., classes, aspects) level by measuring the number of system components used to realize the target concern.

5.4. Concern measurement values

A measured metric value cannot be easily interpreted unless knowing the type and the possible range of values that the metric can assume. To understand how to interpret a measured metric value is fundamental to compare the same metric value with those that are known to be possible values that the metric can theoretically assume. For instance, it is important to know if the value that can be assumed by a given metric has upper/lower limits or not; this let us understand if the measured metric value is high or not.

Criterion instantiation. A set of permissible values may be finite or infinite, bounded or unbounded, discrete or continuous.

Example. In the case, for example, of scattering and size measured in terms of the number of components and operations are bounded and discrete ratio-scale metrics.

5.5. Concern granularity

The granularity of a concern metric is the level of detail at which the measurements are gathered. This criterion is determined by the following factors: (i) element granularity; (ii) element distinction; (iii) direction of the connection; and (iv) aggregation level.

The element granularity factor specifies which elements will be measured, that is, how to aggregate the yielded values. For example, when we say “*the number of concerns of a component that ...*” the entity is a component but what we are counting (granularity) is the number of concerns, i.e., “*concern*”.

The element distinction factor defines how the elements are counted, that is, if we ignore duplicated elements or not when we re-apply the metric to a different goal. For instance, this factor specifies if the same component should be counted for any different concern or not in a given metric.

The direction of the connection factor specifies if the dynamic connections can be in the import direction (i.e., a method is executed on an object call) or in the export direction (i.e., a method is called by another object’s method).

The aggregation level factor specifies the level where the measured information are aggregated. For instance it can be: instance, class or aspect, scenario, use-case, and system.

Criterion instantiation. Possible values for *element granularity* are: (i) concern, (ii) component, (iii) operation, (iv) attribute, (v) member (attribute and/or operation), (vi) lines of code and (vii) bytecode instructions. *Element distinction* has to be “*yes*” (count

⁵ That is, the situation where the execution of one concern triggers the instantiation of many objects from different classes. For example, objects from three classes were instantiated in the execution of CoR in Fig. 1(b).

⁶ That is, the situation where the execution of two or more concerns triggers the instantiation of objects from the same class.

only once) or “no” (count all possible occurrences). Possible values for *direction of the connection* are: “import” and “export” and for *aggregation level* are: “instance”, “component” (class and/or aspect), “scenario”, “use-case”, and “system”.

Example. If we are measuring the strength of the dynamic connection of components realizing a concern, we have to: identify the messages that are exchanged by instances of the system components that are used to realize the concern; aggregate them at the component level; and compute the corresponding metric. In such a case, the entity of measurement is the concern, the unit of measurement is the message while the granularity is the component. Moreover, with the aim of correctly identifying the messages to be counted we have to decide if we have to consider all possible occurrences of the same message independently of its direction or not, i.e., we have to decide if, for example, object “ o_1 uses object o_2 ” corresponds to “ o_2 uses object o_1 ” or if they are two different messages.

5.6. Domain

There are three pertinent issues about domain: (i) the partition of the system to take into account, i.e., application, library or both; (ii) how to account for inheritance, that is how to consider the elements in the hierarchy of the element under analysis; and, in case, (iii) which kind of hierarchy element relationships have to consider.

Regarding the system partition, we have to define which system partition should be accounted for; e.g., the considered elements may belong to the application domain (excluding components of frameworks and libraries used by the system under analysis).

Regarding inheritance, a metric needs to specify if inheritance can be considered or not and which kind of hierarchy element relationships have to be considered. For instance, given a concern metric for run-time coupling among objects; we need to define if messages exchanged with parents of an object must be counted in the coupling metric or not. To precisely define the domain of the measures of interest it is important to limit overestimation and generalization of such measurements when they are actually applied. For example, if we are measuring the size of the components instantiated during the run of a concern, we have to decide how to consider and measure the inherited components. The risk, in fact, is that inherited elements of parent components can be considered twice: when the parent component is instantiated and when a child component is instantiated and used. Hence, different kind of relationships between elements of the element hierarchy need to be considered (e.g., “parents”, “children”).

Criterion instantiation. The possible values for inheritance are “yes” (consider) or “no” (ignore). Besides, if inheritance is taken into consideration metrics have to specify which set of elements should be included: “ancestors”, “parents”, “children”, or “descendants”. Instead, about the domain, we may restrict elements in the domain based on: “application” and “libraries”. Other categorizations are also conceivable.

Example. If we are measuring the size of a concern in terms of number of components that compose it, for example, in an object-oriented system, we have to decide if we want to consider the type hierarchy. For example, if a class A is used to realize the target concern C, and A is child of Pa, to compute the size of the concern C we have two possibilities: (i) only consider A; or (ii) consider both A and its parent Pa. This choice can lead to two different measures especially in large systems. Hence, it is important to precisely define how to compute the concern size. Furthermore, to measure the concern size we have to decide if we have to take into account only the components of the target system or if we have to consider also components used by the application but that are defined into third-party libraries. This

decision can give us more precise or fine-grained information about the size of the software concerns we are measuring, nowadays several software use a lot of third-party libraries to implement their features.

5.7. Concern mapping: concern projection or triggering

One of the most crucial parts in concern measurement is how to project it into elements in the design/application and how to trigger them. At least four issues are related to mapping: (i) what the concerns are, (ii) how the concern can be implemented/exercised, (iii) onto which artifact the concern is going to be mapped, and (iv) how the mapping among concerns and elements can be executed. Clearly, it is not mandatory to specify them. In this case, all kinds of concerns, concern–element maps and concern executions are allowed by the considered metric. Moreover, concern metrics have to specify if they allow concerns overlapping or not. For instance, it is possible that two different concerns could be projected into the same operation. Often, code analysis and inspection are used to map a concern into program/code elements. For instance, static techniques for concept location (e.g., [26]) can be applied to this aim. However, such type of approaches cannot be successfully used to measure dynamic concern properties. Dynamic concern metrics require that the mapping between concerns and (measured) code is realized at run-time (e.g., by taking into account the concern execution). Therefore, at run-time a concern can be triggered by: (i) a system execution; (ii) an executable scenario; (iii) a set of scenarios (use cases). For instance, feature location techniques based on dynamic analysis (e.g., [27]) can be successfully applied to trigger a concern.

Criterion instantiation. As previously explained, a concern can be: a feature, or a set of features, functional or nonfunctional requirements, an implementation mechanism. Possible values for how to exercise a concern can be: feature execution, execution scenarios, use-cases, and system. While, possible values for artifacts can be each computational unit of the system (e.g., components, members, lines of code, code instructions or statements, basic blocks, sub-systems).

Example. Kaur and Johari [7] surveyed existing techniques to map software concern into code elements. Static and semi-automatic techniques are based on the analysis of software artifacts, for instance, the prune dependency rule [28], FEAT [1] and Fan-in Analysis [29]. While dynamic techniques require the definition of execution scenarios (as in [18,19]) that can exercise the concern of interest, thus triggering its code elements, and code instrumentation infrastructures able to capture the triggered elements.

6. Framework evaluation by instantiation

In a first stance, the effectiveness of the proposed framework has been evaluated by formally instantiating a set of dynamic concern metrics. We hence answered the following research question (RQ1):

“Can the framework be used to describe several dynamic and concern-oriented metrics using a common and precise terminology and set of concepts?”

To answer this question, we carried out an experiment with subjects [33] that involved 50 master students enrolled in the software engineering course. The object of the study is the framework and the purpose of the study is to prove that the framework is usable to define a large set of dynamic and concern oriented metrics. The master students attending the software engineering course have a common background in computer science and are acquainted with the necessary concepts as metrics, measurements

and software quality even if they are not experts; in few words they represent the average user: acquainted with the topic but not too skilled. Such a choice has granted a more germane evaluation of the framework avoiding any spike (all perfect or completely a mess) in the feedback.

The students have been asked to use our framework to define and describe the metrics informally presented in Table 1 (plus those defined in [4] and reported in Table 2) and the metrics reported in Table 6 that we did not use to set up the framework. The new metrics used in the experiment (Table 6 third column) are straightforward adaptations of static and module-oriented metrics found in the literature to dynamic concern metrics. The two sets of metrics used in the experiment have different roles. The former set—since used to define the framework—provides an evidence on the usability of the framework: we already know that such metrics could be defined with the framework but we were not aware if this can also be done by people that did not develop the framework. On the other hand, the inclusion of the latter set of metrics in the experiment has permitted to show that the framework is not tailored on the metrics used to define it but it can grasp a wider set of metrics.

The experiment has taken place in two phases. In the first phase the students got acquainted with the framework, thanks to a lecture and a guided tutorial on the framework use we had at the end of the software engineering course. In the second phase each student became an active actor in the experiment by autonomously using the framework to define some of the metrics. In particular in this second phase the students have been scattered all over the examination room with enough space between them to avoid collaborations and then each of them had to draw for two distinct metrics out of a box (the box contained 4 copies of each metric for a total of 100 pieces of paper; the drawn pieces of paper were not reinserted in the box) and then they started to work on the instantiation of the drawn metrics. This phase of the experiment lasted for three hours and it had the final goal to get feedback about the framework usability, completeness and effectiveness. Feedback has been collected through an anonymous form given to the students together with the metrics to be instantiated and the forms are put back in a separate box at the end of the three hours; anonymity permitted to have students' unbiased feedback since it reduces the risk of "teachers' retaliations" in case of negative feedback. Feedback has been collected as open answers to few generic questions like "the framework provides all the concepts necessary to instantiate your metrics? If no, please, explains what it is

missing in your opinion or what do you think it is necessary to change." Basically, the experiment tested if the students were really able to use the framework to instantiate metrics whereas the feedback (together with the hints we could extract from the students' instantiation) were intended to provide suggestion for improvements.

Feedback on the instantiation activity about the metrics in Table 1, confirmed our choices for the framework elements and components (listed in Section 4) and helped to tune up the set of possible instantiations for the framework criteria (listed in Section 5). All the students were able to accomplish their task with a small number of issues and the achieved instantiations were similar to those we realized. Only in few cases the four instantiations of the same metric were discordant. Some of the most interesting issues were:

- (i) the instantiations of the *dynamic operation length* metric revealed that the meaning of *unit* in the framework was ambiguous since some students gave a different interpretation of it (now we extended the framework to cover all possible values for the *unit* element);
- (ii) the instantiation of the *dynamic Concern Sensitive Coupling* metric revealed that the students get confused by what a connection is; in particular the confusion was related to the *direction of the connection* concept (now this concept has been deeply explained) and
- (iii) often the students confused the *unit of measurement* with the *entities of concern measurement*; initially their differences were not clearly written;

the description of these criteria has been rewritten and some examples provided to clearly state the differences.

Apart from what we inferred from the metric instantiations; really few (useful) suggestions came from the students in the answers to the open questions and in general these were related to clarify and improve some framework definitions. For instance, thanks to the students' feedback we have understood more deeply the role of the connections ($CC(S)$) among pairs of system elements like instances and components in different language paradigms (e.g., object-oriented, aspect-oriented and feature-oriented), and we detected the different types of units involved in such paradigms (i.e., classes, components, aspects, features), see Section 4 for details. Feedback on the instantiation activity of the new metrics (Table 6) has permitted to evaluate the completeness of the

Table 6
New concern metrics not considered in the framework construction.

Metric	Original definition	Modified definition
dpubOp	public methods [30] counts the number of "public" methods of a class	dynamic public operations counts the number of "public" operations exercised during the execution of a concern
dOpLength	method length [30] counts the number of methods of a class longer than n (parameter) lines of code	dynamic Operation Length counts the number of operations executed by a concern and that exercise more than n (parameter) lines of code
dDOSC	degree of scattering in components (DOSC) [8] measures the degree to which the components of a system compose a concern	dynamic degree of scattering in components measures the degree to which the components of a system compose a concern at runtime
dDOSO	degree of scattering in methods (DOSM) [8] measures the degree to which the methods of a system compose a concern	dynamic degree of scattering in operations measures the degree to which the methods of a system compose a concern at runtime
dOVERL	concern overlap [31] measures the percentage of overlap of concern code for two or more concerns	dynamic overlap measures the percentage of overlap of concern executed code for two or more concerns
dNso	number of shared operations (Nso) [32] counts the operations of other concerns called by every concern	dynamic number of shared operations counts the operations of other concerns executed by every concern at runtime
diCd	cyclical dependencies (iCd) [32] counts the number of cyclical dependencies of the code elements containing a given concern	dynamic cyclical dependencies counts the number of cyclical dependencies among the components executed by a given concern at runtime
dRR	Reuse (RR) [32] measures the concern reuse in terms of number of inherited concerns per each concern	dynamic reuse measures the concern reuse in terms of number of concerns executed by a concern at runtime
dIC	inner concerns (IC) [32] counts the number of inner concerns of a given concern	dynamic inner concerns counts the number of sub-concerns executed by a concern at runtime
dInC	concerns for a component (InC) [32] counts the number of concerns a given component is participating	dynamic concerns for a component counts the number of concerns that execute a given component at runtime

framework in terms of its elements and criteria and since the students were able to instantiate the whole set without big problems we are confident that our criteria covers a large enough set of possibilities. Note that the version of the framework presented in this paper already takes into consideration all the suggestions for improvements we implicitly (from the instantiated metrics) or explicitly (from students' feedback) got from the experiment. This limited experiment can be considered as the first attempt to validate the framework by means of a set of instantiations, however, further efforts are going to be devoted to widen the experiment and to involve other researchers in this activity for getting an, as much as possible, complete and shared validation of the framework. In any case we consider the experiment satisfactory and it supports our research question RQ1.

Table 7 summarizes the achieved results in terms of values for the most relevant factors of the whole set of instantiated metrics. To complete the picture, we report the instantiation of three of these metrics (basically as done by the students in the experiment): *dynamic Concern Sensitive Coupling (dCSC)*, *dynamic Concern Diffusion over Operations (dCDO)* and *dynamic Degree of Scattering across Components (dDOSC)*. Please note that, dCSC and dCDO (Table 1) have been used to build and refine the framework, while dDOSC (Table 6) is only used to validate the framework and not to define it. By analyzing and selecting each criterion defined by the framework for the chosen metrics we have the following.

dynamic Concern Sensitive Coupling (dCSC) of a concern *con*.

Entity of Concern Measurement. *Concern* is the entity of measurement for this metric.

Attribute. dCSC quantifies *coupling* of each component *c* of the concern *con* ($c \in C(con)$).

Unit. The unit is the *number of (explicit) connections* of the concern components. In other terms, for each component $c \in C(con)$, its explicit connections ($r \in EC(c)$) are considered.

Properties of Values. Permissible values for this metric are not higher than existing (explicit) connections of the system $EC(S)$ (*finite*), do not define any interval a priori (*unbounded*), and allow integers only (*discrete*).

Granularity. The granularity of elements that is being measured is *object* (i.e., the run-time component). The direction of connections is *import*. Only distinct connections are taken into consideration and the level of aggregation is *component*.

Domain. It considers *application components* (not components in the framework or libraries) and *takes inherited operations into account*. In other terms, we consider each component *c* of the application *S* that is part of the concern *con* ($c \in (AC(S) \cap C(con))$). However, it does not count inheritance relationships as connections.

Concern Triggering. Concerns can be a *feature* executed by some input. Concerns are identified by the *execution of scenarios* ($SCN(con)$) that triggers them. *Overlapping* of concerns is allowed.

Using the selected criteria and the concern terminology described in Section 3 we derive the following formal definition for dCSC:

$$dCSC(con) = \{ |r \in EC(c)| \text{ s.t. } c \in (AC(S) \cap C(con)) \wedge \text{exec} \in SCN(con) \wedge con \in Con(S) \}$$

where *exec* is one of the scenarios ($exec \in SCN(con)$) that exercises/triggers the concern *con*; *c* is a component defined in the application

(we are not interested in considering components of libraries and so on) and involved by the concern *con* ($c \in (AC(S) \cap C(con))$) and *r* is an explicit connection from *c* to other components of the system *S*; the cardinality of this set represents the desired value for the metrics.

dynamic Concern Diffusion over Operations (dCDO).

Entity of Concern Measurement. *Concern* is the entity of measurement for this metric.

Attribute. dCDO quantifies *dynamic scattering* of a given concern over the operations of a running system ($o \in Op(S)$).

Unit. The used unit is the *number of operations* *o* of each concern component *c* ($o \in (Op(c) \cap Op(con))$).

Properties of Values. Permissible values for this metric are not higher than $Op(S)$ (*finite*), do not define any interval a priori (*unbounded*), and allow integers only (*discrete*).

Granularity. The granularity of elements that is being measured is *operation* *o*. The direction of connections is *import* (accessing *attributes* or *operations*). *All connections* are taken into consideration and the level of aggregation is *component*.

Domain. It considers *application components* $c \in (AC(S)$ (not components in the framework or libraries) and takes into account *inherited operations from all ancestor components*.

Concern Triggering. Concerns are *features* that can be exercised by the execution scenarios ($SCN(con)$). *Overlapping* of concerns is allowed.

Using the selected criteria and the concern terminology described in Section 3 we derive the following formal definition for dCDO:

$$dCDO(con) = \{ |o \in (Op(c) \cap Op(con))| \text{ s.t. } c \in AC(S) \wedge \text{exec} \in SCN(con) \wedge con \in Con(S) \}$$

where *exec* is one of the scenarios ($exec \in SCN(con)$) that exercises/triggers the concern *con*; *c* is a component defined in the application ($c \in AC(S)$) and *o* is an operation defined by the component *c* and invoked by the concern *con* ($o \in (Op(c) \cap Op(con))$); the cardinality of this set represents the desired value for the metrics.

dynamic Degree of Scattering across Components (dDOSC).

dDOSC is the dynamic counterpart of the original (static) Degree of Scattering across Classes (DOSC) defined by Eaddy et al. [8]. They defined the DOSC metric as the degree to which the concern code is distributed across classes of the system under analysis. The DOSC value of a concern ranges from 0 to 1; when it is equal to 0 all the concern code is concentrated in one class while it is equal to 1 that code is equally subdivided among the classes of the system. The DOSC metric is inspired by—but with a finer grained than—CDC; this metric represents the classes that compose a concern while DOSC represents the degree to which the classes of a system form a given concern. dDOSC for a concern is defined as the degree to which the components of a system are executed by a given concern at runtime.

Entity of Concern Measurement. *Concern* is the entity of measurement for this metric.

Attribute. dDOSC quantifies *dynamic scattering* of a given concern over the system statements ($st \in St(S)$).

Unit. The used unit is the *number of statement* *st* of each concern component *c* ($st \in (St(S) \cap St(con))$).

Properties of Values. Permissible values for this metric are ranging from 0 to 1 (*finite* and *bounded*), and allow decimal value in this range (*continuous*) obtained by dividing integer

Table 7
Instantiation of concerns metrics.

Concern metrics	Entity	Attribute	Unit	Values	Granularity and distinct	Domain and inheritance	Concern, artefact and overlapping
<i>Extended dynamic</i>							
dCBOoC	component	coupling	members	finite, unbounded, discrete	member, yes	application, yes	feature, component, yes
dLCoMoC	component	cohesion	attributes	finite, unbounded, continuous	attributes and operations, yes	application, no	feature, operations and attributes, yes
LCoC	component	size	statement	finite, unbounded, discrete	statement, yes	application, yes	feature, component, no
NToC	component	concurrency	component	finite, unbounded, discrete	component, yes	application, no	feature, component, no
<i>Extended static concern-oriented</i>							
dCD	concern	scattering	components	finite, unbounded, discrete	component, no	application, no	feature, component, yes
dCDC	concern	scattering	components	finite, unbounded, discrete	component, no	application, no	feature, component, yes
dCDO	concern	scattering	operations	finite, unbounded, discrete	operation, no	application, no	feature, operations, yes
dFCD	concern	scattering	components	finite, unbounded, discrete	component, no	application, no	feature, component, yes
dLCC	component	cohesion	concerns	finite, unbounded, discrete	component, no	application, no	feature, component, yes
dSize	concern	size	members	finite, unbounded, discrete	member, yes	application, no	feature, member, no
dSpread	concern	scattering	components	finite, unbounded, discrete	component, no	application, no	feature, component, no
dCSC	concern	coupling	connections	finite, unbounded, discrete	component, yes	application, no	feature, component, yes
<i>Wong et al. [4]</i>							
Concentration	concern, component	closeness	none	infinite, bounded, continuous	member, no	application, no	feature, member, yes
Dedication	concern, component	closeness	none	infinite, bounded, continuous	member, no	application, no	feature, member, yes
Disparity	concern, component	closeness	none	infinite, bounded, continuous	member, no	application, no	feature, member, yes
<i>Extended static and module-oriented</i>							
dDOSC	concern	scattering	statement	finite, bounded, continuous	component, no	application, no	feature, component, yes
dDOSO	concern	scattering	operations	finite, bounded, continuous	operation, no	application, no	feature, component, yes
dOVERL	concern	overlapping	statements	finite, bounded, continuous	statement, no	application, no	feature, component, yes
dNsM	concern	coupling	operations	finite, unbounded, discrete	operation, yes	application, yes	feature, component, yes
diCd	concern	coupling	connections	finite, unbounded, discrete	operation, yes	application, yes	feature, component, no
dRR	concern	tangling	components	finite, unbounded, discrete	component, yes	application, no	feature, component, no
dIC	concern	reusing	concern	finite, bounded, discrete	component, yes	application, no	feature, component, yes
dInC	concern	tangling	concern	finite, bounded, discrete	component, yes	application, yes	feature, component, yes
dOplength	concern	complexity	statements	finite, unbounded, discrete	operation, yes	application, yes	feature, component, no
dpubOp	concern	complexity	operations	finite, unbounded, discrete	operation, yes	application, yes	feature, component, no

values representing number of system statement st (i.e., not higher than $St(S)$).

Granularity. The granularity of elements that is being measured is *statement* st . The direction of connections is *import* (accessing *statements*). All *connections* are taken into consideration and the level of aggregation is *component*.

Domain. It considers statements of *application components* $c \in (AC(S))$ (not components in libraries) and takes into account *inherited operations from all ancestor components, containing statements*.

Concern Triggering. Concerns are *features* that can be exercised by the execution scenarios ($SCN(con)$). *Overlapping* of concerns is allowed.

Using the selected criteria and the concern terminology described in Section 3 we derive the following formal definition for dDOSC:

$$dDOSC(con) = \left\{ \frac{|\{st \in (St(c) \cap St(con))\}|}{|\{st \in St(con)\}|} \text{ s.t. } c \in AC(S) \wedge \text{exec} \in SCN(con) \wedge con \in Con(S) \right\}$$

where $exec$ is one of the scenarios ($exec \in SCN(con)$) that exercises/triggers the concern con ; c is a component defined in the application ($c \in AC(S)$) and st is a statement defined by the component c and invoked by the concern con ($st \in (St(c) \cap St(con))$); the cardinality of this set divided by the cardinality of the concern statements ($st \in St(con)$) represents the desired value for the metrics.

Overall remarks. Summarizing, the experiment showed that we can positively answer to the research question RQ1, i.e., the framework is usable to describe a large set of concern-oriented metrics both static and dynamic.

7. Dynamic concern-oriented metrics for bug-proneness

In this section we report on a case study carried out to provide an initial evidence of dynamic concern metrics utility and effectiveness. Despite their potential usefulness, in fact, the use of such metrics is scarcely investigated in the existing literature (cf. Section 8). The study hence shows how dynamic concern measurements—in particular, size and scattering—(1) can be calculated in actual cases; and (2) how they can be exploited to predict bug-proneness of application code.

In this case study we tried to address the following research question (RQ2):

“Are the dynamic concern-oriented metrics useful to predict the concern bug-proneness?”

In particular, according to the existing literature and by considering that:

- i. several software characteristics and properties (e.g., software size, complexity, coupling, scattering degree) can potentially contribute to the bug-proneness [28,34] of a system;
- ii. both static and dynamic characteristics and aspects (e.g., static coupling vs. dynamic coupling) can impact on the bug-proneness of a software system [35–37] and that
- iii. static and dynamic metrics can have comparable behavior and trends when they are measuring related software properties and characteristics, and/or they complement each other, when they are measuring unrelated properties [38,39].

We can expect that static and dynamic concern metrics can complement each other in evaluating and predicting the concern defectiveness. We investigate this intuition in the case study.

Table 8
Statistics about the considered applications (missing data are unavailable).

Application	Lines of code (LOCs)	# developers	Downloads			Bugs	
			# from 2000	# last week	Year of first one	# bugs	Density (%)
Mtac ^a	11k	2	1695	1	2003	21	0.19
Buddi ^b	18k	1	1,014,173	926	2006	279	1.55
jMove ^c	40k	3	4357	1	2002	53	0.13
JTopas ^d	2k	1	9606	5	2001	13	0.65
XmlSecurity ^e	43k	–	–	–	–	345	0.8
DbViz ^f	6k	4	15,812	12	2002	69	1.15
Jtidy ^g	18k	8	314,071	303	2000	176	0.97

^a <http://sourceforge.net/projects/mtac>.

^b <http://buddi.digitalcave.ca>.

^c <http://jmove.sourceforge.net>.

^d <http://jtopas.sourceforge.net/jtopas>.

^e <http://santuario.apache.org>.

^f <http://jdbv.sourceforge.net/dbViz>.

^g <http://jtidy.sourceforge.net>.

7.1. Case study analysis

The case study considers seven Java applications summarized in Table 8. All of them are open-source systems and their source code, JUnit test cases, software documentation and bug trackers are available through their websites. The study covers a large variety of applications in terms of code lines (size) and number of downloads (diffusion). Buddi, Jtidy and DbViz are successful applications (high downloads rate) while Mtac and jMove have a quite limited number of downloads. Since the users play also the role of testers in open-source applications, a low number of users reflects on a low number of bugs detected; e.g., jMove is quite large (40 k LOCs) but only few bugs (53) have been detected that can be explained by the low number of downloads/users (only 4357 since 2002).

Our case study is inspired by the one presented by Eaddy et al. in [28] but it has different context, i.e., objects of the study as well as tools supporting the subject of studies in collecting concern measurements are different, and goals, i.e., Eaddy et al.'s goal was to capture the relationships between static concern measures and the concern defectiveness while we aim at studying the role of dynamic concern measures on the concern bug-proneness prediction. In the Eaddy et al.'s case study, in fact, some software systems have been analyzed looking for the relationships between static concern measures—i.e., size and scattering degree—and the concern defectiveness—that is the number of defects in the concern. From the study, Eaddy et al. [28] observed that concern size and scattering degree have a negative impact on the concern defectiveness, that is an increment of the concern size and/or scattering degree implies an increment in its error-proneness. Similarly to Eaddy et al.'s study, we measured size and scattering degree of a set of software concerns for the considered Java applications. Differently from them, however, we used both static and dynamic counterparts of three of the presented metrics (Section 6) to point out how concern-oriented metrics can be used in software bug-proneness and the difference, if any, between static and dynamic measurements.

Our case study consisted of five steps: *concern selection*, *concern mining*, *concern defectiveness*, *concern measurement* and *correlation analysis*; the whole process is repeated for each considered application.

7.2. Concern selection

This activity aims at identifying the set of concerns that have to be analyzed for each considered application. To select a set

of suitable concerns, the application's functional requirements are identified and analyzed since each application requirement represents a concern that is a potential candidate for our study. For example, in the case of a software that simulates ATM services, a functional requirement (i.e., a concern candidates for our study) could be the functionality enabling the customer of making a deposit to a given bank account. Conversely, any non-functional requirements, e.g., software maintainability or security, is not a valid candidate for our study. Hence, we first analyzed the application requirements documentation and, more frequently, the user manuals and the application web site for identifying the provided functionality provided. We then considered only those functionality whose code represents a non trivial concern. Overall, the set of selected concerns should cover almost the main functionality provided by the considered application with a minimal overlapping. A concern C_{over} overlaps another concern C if C_{over} is a sub-concern of C , e.g., all program elements of C_{over} are part of C [28].

7.3. Concern mining

Each selected concern is mapped to the corresponding portion of code. This process depends on the kind of performed measurements: static or dynamic. In the former case, static code analysis and the *dependency rule*⁷ are used to point out the concerns. In the latter case, concerns are triggered and pointed out by executing the test cases associated to the concerns—the study exploited the JUnit test cases provided with the application. In the case study this step has been (partially) automated by using FLAT³⁸ [40], JRipples⁹ [41] and AOP➔HiddenMetrics [20].

7.4. Concern defectiveness

The actual defectiveness (i.e., number of bugs) of each considered concern has been determined by manually inspecting the application's bug tracker, on-line documentation and code repository. By looking at the bug trackers, we identified all failures reported by the application users (e.g., report of a crash, report of a functionality not implemented correctly) and recognized as actual failures, thus solved and closed by the application developers. For each of these reported failures, we again analyzed both bug

⁷ A program element (class, method and code statement) is related to a concern if a dependency exists, e.g., the program element is removed when the concern is removed or changed [28].

⁸ <http://www.cs.wm.edu/semeru/flat3>.

⁹ <http://jripples.sourceforge.net>.

tracker and code repository with the aim of identifying the bug/s (e.g., a method called with wrong parameters values, un-initialized variables) that cause/s the reported failure. To this aim, we looked in particular at the code patch implemented by the developers to fix the bug, thus solving the failure. The failure was discarded, in case: (i) it was a failure related to a software version different from the one considered in the experiment, (ii) it was a still open (i.e., not solved) failure, (iii) it was not an actual failure but for instance the request of a new software feature, (iv) we were not able to identify the bug causing it, and (v) it was an unclear or duplicated failure description. According to [28], each bug was then mapped to the concern/s where it occurred to determine which is the most buggy concern. The bug was discarded if it occurs in a non-considered concern.

7.5. Concern measurement

Size and scattering degree have been statically and dynamically measured for each selected concern. The concern size has been calculated as the number of code lines realizing the concern (SLOCs and dLOCs) while the scattering degree has been calculated considering the concern diffusion over components (sCDC/dCDC) and operations (sCDO/dCDO). See Tables 1–6 for the metrics definition.

7.6. Correlation analysis

The correlation between the measured metrics and the actual concern defectiveness has been initially evaluated by applying the *Spearman's correlation coefficient* [42] to determine its existence and strength. Then, a multiple regression analysis—Nelder et al. [43]'s generalized linear model—has been performed to bind the measured metrics to the variance of concern defectiveness and to determine their capability as predictors of the actual concern defectiveness. The goal of this analysis is hence threefold: (i) answering the research question RQ2 about the usefulness of concern-level metrics; (ii) identifying the relationships between concern properties (as measured by the metrics) and defectiveness, if any; and (iii) understanding the impact of dynamic metrics with respect to their static counterpart in the prediction of concern defectiveness.

7.7. Case study results

In the next, the study's results are summarized step-by-step.

7.8. Concern selection

Twenty-eight concerns—reported in Table 9—have been selected from the functional requirements of the 7 considered applications. These concerns represent all the relevant functionality provided by the applications with a limited overlap. For instance, `ExpressionCalculator` is the most relevant feature of `Mtac`, it provides in fact the capability of executing mathematical operations while `PlotManager` is controlling the user interface, that is collecting the input values then elaborated by `ExpressionCalculator` and showing to the user the output produced by the same `ExpressionCalculator`.

7.9. Concern mining

How the concern/code mapping has been realized depends on the type of measured concern metric. If the metric to measure is static the mapping is based on static code analysis to determine the code element dependencies and on the application of the *dependency rule* to determine the relevance of each code element for the concern under analysis.

Fig. 2(a) shows an Eclipse screenshot of the environment used to perform such an analysis for the `ReportGenerator` concern of the `Jtidy` application. First, `FLAT`³ has been used to determine a textual similarity between the concern description extracted from the application requirements and the source code elements. The result was a list of few core code elements for each concern, e.g., in the bottom left pane in Fig. 2(a) are listed the elements for the `ReportGenerator` concern. This list was passed to `jRipples` to identify new dependencies and to evaluate—thanks to the application of the *dependency rule*—the relevance of each element for the concern and therefore which element implements it. The right pane in Fig. 2(a) shows the classes composing the concern under analysis.

Conversely, if the metric to measure is dynamic, the `jUnit` test suite provided with each application is analyzed and each test case is associated to the concern it mainly triggers thanks to the tool for code coverage `Emma`¹⁰; then the test cases associated to the concern under analysis are executed by means of our tool `AOPHiddenMetrics` [20] to apply the dynamic metric measurement to the concern. Fig. 2(b) shows, as an example, the code coverage information for the `ReportGenerator` for `Jtidy`; note that `ReportTest.java` is the unit test provided with the application triggering the homonymous concern.

Overall, to map a concern into the code is a crucial and time consuming task but, as you can see:

Mapping time	Mtac	Buddi	jMove	Jtopas	XMLsec	DBViz	Jtidy
Static concerns	1h06'	3h35'	6h45'	2h10'	5h30'	1h45'	3h15'
Dynamic concerns	15'	25'	1h20'	40'	1h40'	55'	27'

the process is heavier (between 2 and 6 times) in the case of static concern measurements. As we will observe in our analysis (see below), this is mainly due to the semi-manual analysis conducted to statically map concern to the code versus the automatic analysis conducted for dynamically map concern to code.

7.10. Concern defectiveness

Given the mapping between concerns and code it has been possible to associate the bugs in the application bug tracker to the considered concerns:

	Mtac	Buddi	jMove	Jtopas	XMLsec	DBViz	Jtidy
Considered (tracked) bugs	16 (21)	96 (279)	31 (53)	16 (16)	160 (345)	24 (69)	36 (176)

In the study, 379 bugs have been selected among those we were able to find in the bug trackers (959 bugs) of the considered applications. 580 bugs were discarded because: (i) they were related to a software version different from the one we considered: 40%, (ii) they were still open bugs: 8%, (iii) they were not actual bugs but they were requests of new features: 16%, (iv) we were not able to identify the bug causing it: 21%, and (v) their description in the bug tracker were confused, unclear or duplicated: 15%. The 379 considered bugs have been then distributed among the 28 considered concerns; the distribution is reported in the last column of Table 9. This has permitted to determine the most buggy concern

¹⁰ <http://emma.sourceforge.net>.

Table 9
Considered concerns and measured metrics per concern.

Application	Concern	Dynamic			Static			Bug
		dLOCs	dCDC	dCDO	sLOCs	sCDC	sCDO	
Mtac	ExprssionCalculator	870	47	148	3645	55	212	7
Mtac	PlotManager	640	26	133	1602	26	189	9
Buddi	Account	645	33	158	3728	80	655	26
Buddi	Budget	700	32	170	2915	78	700	23
Buddi	Transaction	883	40	186	2409	80	655	31
Buddi	Reports	546	31	146	3253	85	693	16
jMove	CodeAnalysis	7079	100	1241	14,866	138	1365	7
jMove	Dependencies	7073	101	1242	14,924	140	1367	8
jMove	Metrics	7158	102	1255	15,071	143	1384	9
jMove	Statistics	7283	106	1298	30,576	291	2807	7
Jtopas	Tokenizer	509	8	102	2309	26	281	6
Jtopas	Plugin	323	5	76	1705	14	187	1
Jtopas	InputStream	542	6	90	1496	9	148	6
Jtopas	TokenProperties	452	6	79	1496	9	148	3
XMLsec	Cl4Helper	537	35	84	5737	49	451	16
XMLsec	Canonicalizer	1002	38	129	6502	42	479	33
XMLsec	XalanBug	559	40	90	7400	47	598	12
XMLsec	InteroperabilityBaltimore	2298	65	380	4777	26	395	67
XMLsec	XMLSignature	737	49	111	8888	92	922	32
DBViz	importSQL	673	23	113	590	10	39	10
DBViz	PrintDiagram	727	22	113	326	7	18	5
DBViz	StartDBviz	447	29	75	169	2	11	2
DBViz	InputSchema	599	32	91	667	10	41	7
Jtidy	Configuration	1291	57	259	5096	62	115	7
Jtidy	Lexer	50	3	10	3900	9	90	12
Jtidy	ReportGenerator	582	20	82	1728	20	106	4
Jtidy	Utility	245	3	11	2741	39	115	3
Jtidy	Encoding	220	2	5	1341	5	59	10

for each application, e.g., `PlotManager`, `Transaction` and `Lexer` resulted the most buggy concerns for `Mtac`, `Buddy`, and `Jtidy` respectively.

7.11. Concern measurement

To permit a germane interpretation of the results, our study considers exactly the same metrics (lines of code—LOCs—, concern diffusion over components—CDC—and concern diffusion over operations—CDO) measured both statically and dynamically. [Table 9](#) reports the measured size and scattering degree for each considered concern. In general, we can observe that static measurements tend to be more conservative and their variability in different concerns is higher (i.e., a higher median and standard deviation) than for the dynamic measurements. With respect to the single metrics instead we can observe that CDC seems to be, on average, the most conservative metric in both static and dynamic versions (*low variability*), while, LOCs largely variates when statically or dynamically measured (*great variability*).

7.12. Correlation analysis

The correlation between two variables reflects the degree to which the variables are related. The *Spearman's correlation coefficient* (ρ) is the most common measure of correlation and reflects the degree of linear and non-linear relationship between two variables.¹¹ By applying the Spearman's correlation to pairs of static,

¹¹ The correlation ρ ranges from +1 to -1, the value +1 means that there is a perfect positive linear relationship; while a coefficient of 0 means no correlation. The p -value supporting the ρ correlation value is the probability that one would have found the current result if the correlation coefficient ρ were in fact zero (i.e., null hypothesis); if this probability is lower than the 5% the correlation coefficient is called *statistically significant*.

Table 10

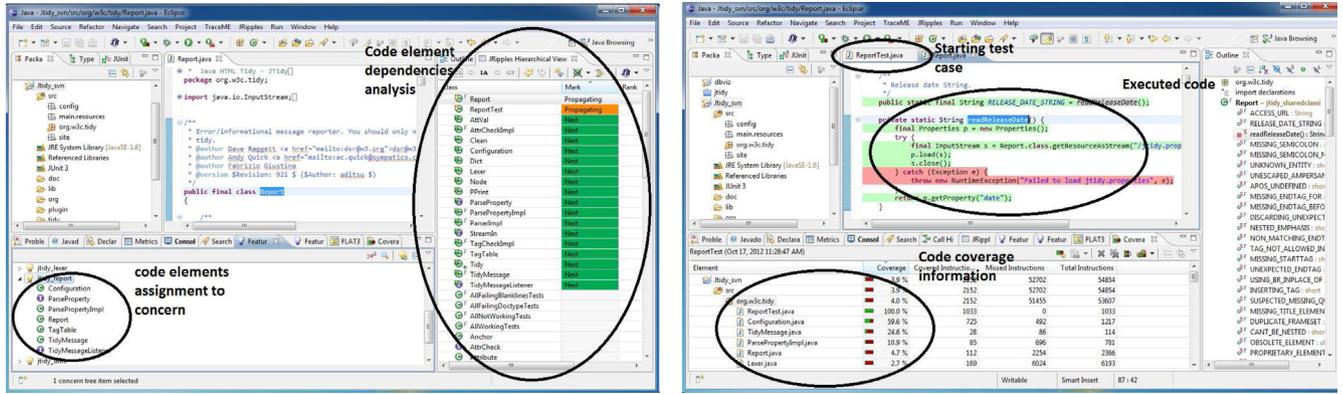
Spearman's correlation coefficient. Notice that the correlation between dLOCs and Bug is statistically significant at 10% (i.e., p -value < 0.1) while the others at 5% (i.e., p -value < 0.05).

	dLOCs	dCDC	dCDO	sLOCs	sCDC	sCDO	Bug
dLOCs	x	0.88	0.92	0.58	0.69	0.59	0.33
dCDC		x	0.84	0.74	0.78	0.7	0.40
dCDO			x	0.57	0.76	0.69	0.36
sLOCs				x	0.82	0.81	0.44
sCDC					x	0.9	0.40
sCDO						x	0.46

dynamic and static-dynamic metrics (the results are listed in the first six columns of [Table 10](#)), we observed that: (i) there is a strong¹² correlation between pairs of static or dynamic metrics (e.g., the correlation between dLOCs and dCDC is $\rho = 0.88$, with p -value < 0.05); and (ii) there is a moderate to strong correlation between pair of static and dynamic related to the considered concern properties (e.g., the correlation between dLOCs and sLOCs is $\rho = 0.58$, with p -value < 0.05). Such a result depends on having considered properties (size and scattering) of the same “strongly related” set of code elements; in other terms, an increment on the number of classes implies an increment of methods and lines of code.

On the other hand, the measured metrics seem to moderately correlate with the number of bugs associated to the concerns—as reported in the last column of [Table 10](#). Moreover, we see that the static metrics have, on average, a better correlation with bugs than their dynamic counterparts. These correlation values are

¹² In the paper the observed correlation is described using the scale proposed in [44].



(a) Environment for the static analysis

(b) Code coverage information

Fig. 2. Screenshots from the Jtidy example of concern-code mapping.

aligned with Eaddy et al. [28]’s results and they depend on dominating factors effect as well as on the existence of un-expected confounding factors (those factors, e.g., the programming language of the analyzed application, that even if it is not identified can potentially influence both the measured concern metrics and the concern defectiveness).

Then the stepwise regression analysis [45] has been applied to discover and measure these dominating factors (i.e., metrics), if any. The correlation quantifies the degree to which a pair of variables is related, thus giving an idea about how much one of the variable tends to change when the other one changes. However, the existence of correlation does not imply causation. For instance, a third (unobserved) variable related to both investigated variables can exist and can be the actual responsible for the observed correlation. In this view, the regression analysis measures the proportion of variability explained by or due to the regression relationship between the variables under consideration, i.e., the objective is to predict values of a variable based on values of other ones.

The stepwise regression analysis is commonly applied to reduce a set of variables (*regression model*) according to their correlation degree. To correlate concern properties with defectiveness, a stepwise multiple regression analysis that considers all the measured metrics per considered concern has been applied. The model has been progressively reduced by removing those metrics with a lower correlation degree. The resulting model¹³ contains 5 out of 6 considered metrics: sLOCs, sCDC, sCDO, dCDC, and dCDO. Table 11 summarizes the built model, the probability of each metric of effecting on the concern defectiveness (the column labeled «Estimate») and the corresponding *p*-value for the statistical significance analysis (the column labeled «Pr(>|t|)»).

Table 11 shows that concern size (both statically and dynamically measured) does not significantly effect the concern defectiveness and the *p*-value of sLOCs reveals that its impact is not statistically significant. This result can depend on the low bug density reported for Mtac and jMove. Conversely, Table 11 shows that concern scattering degree (CDC and CDO in both static and dynamic variants) impacts on the concern defectiveness, i.e., scattering metrics explain some of the variance in the number of bugs per concern. Even if sCDO and dCDO have a quite limited effect on the defectiveness, sCDC and dCDC have a strong effect on it. In particular, sCDC results to have a negative effect, i.e., an increase of such metrics would result in a decrease of the probability of having bugs in the concern, while dCDC a positive one, i.e., an increase of

Table 11

Stepwise regression model, with [47]’s pseudo $R^2 = .52$ and AIC = 219.5.

Coefficients	Estimate	Std. error	t value	Pr(> t)
(Intercept)	3.4	3.8	0.9	0.37
sLOCs	-0.001	0.001	-1.4	0.16
sCDC	-0.39	0.15	-2.5	0.018
dCDC	0.69	0.16	4.1	0.0004
sCDO	0.057	0.018	3.1	0.0049
dCDO	-0.049	0.013	-3.5	0.0018

dCDC would increase the probability of having bugs in the concern. An increase of the dCDC metric represents an increase of the concern size, scattering and complexity, thus an increase of concern’s bug-proneness. Conversely, an increase of the sCDC could not represent an actual increase of the concern size, scattering and complexity. This should depend on the over-approximation of the code mapped to the concerns for static metrics with respect to the actual code of the concerns, i.e., the concern mining step could result in a code fragment larger than the effective code of the considered concerns in the case of static code analysis. Hence, this over-approximation could explain the different impact we observed between the static and dynamic metric to the bug-proneness. However, only further experimentation could support this argumentation.

From our stepwise regression analysis, we can observe that concern size and scattering degree have a limited to strong impact in explaining the variance of the concern defectiveness (this result is consistent with the ones of Eaddy et al. [28]). In fact, Table 11 shows that most of the dynamic metrics we considered survive in the final model, i.e., they impact on the bug-proneness as well as their static counterpart. Furthermore, from the table we also see that static and dynamic metrics complement each other in explaining the observed variance. For instance, as said we see that sCDC results to have a negative effect but dCDC has a positive one. The results achieved by static analysis and measurements tend to be over-generalized since static analysis is conservative while the results achieved by applying dynamic analysis and measurements tend to be under-generalized since dynamic analysis consider a limited set of software behaviors. Hence, we guess that by having in the final model (Table 11) both static and dynamic metrics, we can limit the intrinsic characteristics and limits of static and dynamic analysis and measurements. That is, for example, considering dynamic scattering we can limit the impact on the defectiveness prediction of the over-generalization caused by the adoption of (only) static analysis and metrics.

¹³ That is, the one with the minimal Akaike information criterion (AIC); the AIC is a measure of the relative goodness of fit of a statistical regression model [46].

Table 12
Prediction accuracy.

	Full regression model	Static metrics	Dynamic metrics
<i>Prediction accuracy based on the regression model</i>			
MMRE	1.4	1.8	1.08
MdMRE	0.71	0.77	0.5
sd_{MRE}	2	3	1.6
$Pred_{25}$	0.78	0.71	0.78
$Pred_{50}$	0.89	0.82	0.92

As further analysis, we used the built regression model as a proxy (indicator) to predict the concern defectiveness. We built a prediction model by considering the output of the regression analysis. In particular, we consider here three models built: using all metrics, only static and dynamic metrics. By applying the leave-one-out cross-validation to the output produced by the model we obtained the prediction accuracy values summarized in Table 12. The accuracy gives us an idea about the prediction system performance (“goodness” of estimations) of the metrics of being indicators of the concern defectiveness. The accuracy [48] of prediction systems are often measured in terms of magnitude relative error (MRE) and by counting the number of predictions within $m\%$ of the actual values (often m corresponds to 25% or 50%). Table 12 hence summarizes the result achieved by the built predictor model in terms of MMRE (mean MRE), MdMRE (median MRE), $Pred_{25}$ and $Pred_{50}$ as well as standard deviation of MRE (sd_{MRE}). The obtained MMRE and MdMRE values (the results of the full regression model is shown in Table 12 second column) denote that some estimated concern defectiveness values are a bit far from the actual concern defectiveness (slightly more than the double of the actual value), but the result of $Pred$ shows that 78% and 89% of the estimated defectiveness are respectively around the 25% and 50% of the actual defectiveness value. This analysis confirms again that concern size and scattering degree can be used as proxy (indicators) of the concern defectiveness, indeed the models achieved a reasonably accuracy while limiting the error rate. By comparing the results of the full regression model with those achieved by the models built using only the static (Table 12 third column) or the dynamic (Table 12 fourth column) metrics it is possible to see that the regression models built using only dynamic metrics has a better performance, thus contributing to the improvement of the full model.

7.13. Overall remarks

Summarizing, this case study shows that research question RQ2 can be positively answered, i.e., dynamic concern-oriented metrics are useful in bug-proneness prediction tasks. We observed, in fact, that dynamic metrics complement static metrics in predicting the bug-proneness of concerns, e.g., by limit the over-generalization introduced by the adoption of static analysis and measurements. In detail, we observed that static metrics correlate with the concern defectiveness slightly better than dynamic ones (Table 10), but both static and dynamic metrics have a significant impact in explaining and predicting the concern defectiveness (Tables 11 and 12). Table 12 (third column) shows that static metrics can be used to achieve a reasonable accuracy result on predicting the concern defectiveness but Table 12 (second column) shows also that by using both static and dynamic metrics we can improve the achieved results. In our opinion, this can be mainly due to the presence of unused code (e.g., “dead” code or code of other concerns), that is, the so called over-generalization. However, to statically map each concern into the application code is not an easy but rather an error-prone task and the resulting mapping can be too coarse-grained for the actual concern code (e.g., it can contain

program statements or classes not-related to the current concern). In the study, the static measurements required, on average, 3 times the effort (i.e., time spent) devoted to perform dynamic measurements, due to the semi-manual versus automatic concern code mining applied in the two cases, and it resulted in a less accurate mapping. Hence tools supporting the developers in the static concern code mining should help to improve the results. In the meantime, using both static and dynamic metrics can overcome such a limitation of the static metrics, thus let us achieve better results in evaluating and predicting the bug-proneness of the concerns.

7.14. Threat to validity

Unfortunately, the obtained results cannot be easily generalized to other applications and situations since there are several threats to validity. As always happens for case studies, in fact, only repetitions by other researchers and considering other context (object and subject of the study) can better support the achieved results. In the rest of this section we will discuss the prominent threats affecting the validity of our study and we classified them according to Wohlin et al. [33]:

- i. threats to construct validity: threats concerning the relationship between theory and observation;
- ii. threats to internal validity: threats impacting the actual causes of the study outcome;
- iii. threats to external validity: threats which limit the ability to generalize the obtained results;
- iv. threats to conclusion validity: threats concerning the relationship between the main treatment considered in the study and study outcome.

First, the subjectivity degree that affects some tasks of the study could limit the study validity (*internal validity*). For instance, the selection of the concerns used in the study is a quite subjective task and this subjectivity could negatively influence the results. We tried to limit the subjectivity by applying a set of predefined criteria and considering a not trivial number of concerns selected from different applications of different domains.

Again, the subjectivity degree that affects the mapping between concern and code (*internal validity*) and the limited set of considered applications and concerns per application (*external validity*) threaten the study validity. Also the granularity (e.g., classes, fields and methods wrt lines of code) of the concern mapping can affect the result (*internal validity*). We are aware that different or wrong mapping could negatively affect the achieved results (*construct validity*). For example, the lack of links between code elements and concerns could lead to under-generalization of the code realizing a given feature or concern as well as too coarse-grain granularity in this mapping activity could lead to an over-generalization of the code that realizes a concern. Both under/over-generalization can alter the achieved results. However, we are also aware that such a threat to validity cannot be completely eliminated, more research is required toward the mapping of a feature/concern to a piece of code. To limit such a threat, however, we tried to analyze the application tracker and code repository, to automate the mapping task as much as possible by using tools supporting it, to adopt tools, mapping processes and criteria (e.g., the prune dependency rules) successfully used by Eaddy et al. [28] in their experiments.

Another threat to the study validity is due to the used test suites (*internal validity*), we are aware that different test suites could lead to potentially different results. In fact, different test suites can have a different bug finding capability and can be differently mapped to application concerns. To limit such a threat we used actual test suite distributed with the application code by developers, thus

used to test the application for finding bugs and for checking the fixed code, i.e., after code maintenance activities.

Another threat to the study validity regards the used sets of bugs (*internal validity*), they can potentially threaten the study validity. Indeed, different set of bugs affecting the applications, in fact, could potentially influence the achieved results since they impact on different set of concerns (i.e., functional requirements of the application under analysis) and they can be revealed by different test suites. Moreover, different sets of bugs can be differently mapped in the application code, thus in the application concerns (*construct validity*). To limit such a threat we used actual bugs described in the application bug trackers and solved by the application developers by means of code patches.

Other threats to the study validity concern the applications used as objects of our study. We used 7 open-source applications developed in Java (*internal validity*). We used a not trivial but, however, limited number of applications so further repetitions of the study could extend the number of the considered applications. We tried to select applications having different characteristics (different domain, different size and complexity), thus we consider the select ones quite representative of the existing open-source applications developed in Java. Another threat to validity (*external validity*) that regards the application under analysis in this study are the fact that such applications are small to medium applications (their size in terms of lines of code ranges from 2 k to 43 k). We consider such applications quite representative of the medium-size applications typically distributed in the open-source community. Repetitions of the study should involve bigger and complex applications. As already said, all the considered applications are developed in the same programming language (Java), thus we cannot extend the achieved results to applications written by using different programming languages. Again, all these application are open-source applications. On one side, this was useful to have access to several application artifacts (e.g., code, test cases, documentation, bug trackers) but, on the other side, we cannot extend the achieved results to commercial application. Only additional studies can consider commercial applications, even if, we have to say that, it is always difficult to have access to their artifacts.

Finally, to limit the treats to validity related to the relationship between treatment and outcome (*conclusion validity*), we conducted statistical analysis on the collected data by using the Spearman's correlation coefficient and the multiple regression analysis. By means of this kind of analysis we derived conclusions and answered to the research question of interest to the study, thus we limited the subjective interpretation and analysis of the collected data.

Nevertheless these threats to validity, in this case study we observed that by considering also dynamic metrics at concern-level we increased the accuracy of the conducted software analysis and we decreased the effort required to obtain such a prediction. Therefore, our case study encourages us in considering dynamic metrics useful to measure additional dimensions of software properties also at concern-level. Further investigation will be required to a large benchmark of software systems and concerns.

8. Related works

Several works in the literature discuss metrics-based frameworks: Table 13 gives an overview of the state-of-the-art in this field. We grouped the existing frameworks according to their objective and the granularity of the metrics they investigated. In detail, we identified three main objectives: (i) *definition* about definition and description of software metrics; (ii) *use and interpretation* about the use and interpretation of existing software metrics; and (iii) *validation* about the validation of software metrics.

Table 13
Overview of the existing metrics frameworks.

<i>Metrics framework</i>			
Definition			
Module		Concern	
Static	Dynamic	Static	Dynamic
[49–54]	[35,53,10–13,36]	[55,8,17,32,14]	[4]
Use and interpretation			
Module		Concern	
[30,56,57,54,11,50,58,59]	[60,11,13]	[61,62,28,14]	
Validation			
Empirical		Theoretical	
[63,14,64,59,34,65]		[53,51,65–67]	

The frameworks can concern, in fact, metrics working at module-level i.e., isolated piece of code grouped in units (e.g., programs, files, and classes) and at concern- or feature-level (piece of code crosscutting several units). Furthermore, frameworks have been grouped according to the type of validation they proposed: empirical rather than the purely theoretical one. In the rest of this section, we summarize the works in Table 13 by detailing some of the representative ones.

8.1. Metrics definition

Most of the efforts have been spent to construct frameworks for defining design-time and object-oriented metrics devoting to estimate static and/or dynamic properties of an application. For instance, Jacquet et al. [51] detailed step-by-step the process to be applied for defining new metrics while Arisholm et al. [10] presented a measurement framework composed of dynamic metrics for object-oriented systems. Recently, some frameworks for defining metrics for investigating properties of an application at concern-level have been proposed. In particular, almost all efforts on this subject have been spent to investigate static properties. For instance, Eaddy et al. [8] and Sant'Anna et al. [14] proposed two frameworks specifically devoted to analyze software concerns. They introduced, for example, metrics for evaluating how much some concerns are scattering and tangling on the original code in which they are implemented. While for what concerns dynamic metrics, as already explained in the paper, Wong et al. [4] seems to be the unique relevant work.

8.2. Metrics use and interpretation

Frameworks guiding the users in the use and interpretation of metrics are fundamental to be able to apply metrics in practice. For instance, Erni et al. [30] suggested the use of a three-layer (factor-criteria-metrics) quality model that relates several metrics to a number of structural measurements to design principles and rules, aiming at evaluating and estimating the software system quality. Eaddy et al. [28], as stated before, empirically proved that there exists a relationship between some static concern properties (e.g., size and scattering degree) and the concern error-proneness.

8.3. Metrics validation

One of the major threat to validity limiting the use of new metrics concerns their validation. Several frameworks have been presented to empirically and theoretically validate module-oriented metrics. For instance, Briand et al. [67] presented a framework then used to empirically investigate a suite of object-oriented design

metrics as quality indicators. They empirically proved the capability of some object-oriented metrics to predict class fault-proneness. Moreover, Basili et al. [34] proposed a generic and rigorous mathematical framework composed of terms and notions characterizing several software measurement properties (e.g., size, length, complexity, cohesion, coupling). The framework is hence proposed to conduct theoretical validations of software metrics, according to the properties that such metrics measure.

Differently to all these kind of works, we tried to fill a gap of the literature proposing, on one side, a new framework composed of terms and notions that can be used to describe the existing metrics while, on the other side, we also introduced and empirically evaluated some new dynamic concern-oriented metrics.

9. Conclusions

In this paper we have presented a framework to define and describe software metrics for measuring dynamic properties of applications at concern level. This framework extends the one presented in [9] that was limited to define concern-oriented metrics for measuring (only) static properties of a system. This extension permits to describe software metrics for measuring dynamic properties of a system as well. The result of the work is a new framework that introduces a unified terminology and a set of criteria used in a consistent and rigorous process to define well-founded (static and dynamic) concern-oriented metrics for aspect- and component-oriented applications.

To answer to the following question (RQ1 in Section 6): “Can the framework be used to describe several concern-oriented metrics using a common and precise terminology and set of concepts?” we conducted an experiment in which we have instantiated several existing and new concern-oriented metrics by applying our framework. This experiment helped us to improve and complete the framework and showed that the framework could be used to describe a wide range of concern metrics.

In the paper, we reported also a case study conducted to answer the following question (RQ2 in Section 7): “Are the dynamic concern-oriented metrics useful to predict the concern bug-proneness?”, thus giving to the reader an idea about both the utility and the effectiveness of the dynamic concern metrics. In the study, we measured the capability of some static and dynamic concern metrics in evaluating and predicting the concern bug-proneness. Even if quite preliminary, the achieved results of this study show that dynamic metrics improve the quality of prediction systems. Further experimentation and case study repetitions could support our findings.

In the future, we plan to extend our case study analysis for different kinds of applications: object-, aspect-, and component-oriented. The main aim of this work will be to understand the real impact of both static and dynamic software concern-measurements and their real effectiveness when applied to support software development and maintenance.

Acknowledgements

The authors wish to thank Alessandro Garcia, Eduardo Figueiredo and Thiago Bartolomei for their precious help on defining the static version of this framework. Moreover, the authors wish to thank the anonymous reviewers for their insights that helped to improve the scientific content and presentation of this work. Walter Cazzola’s work has been partially supported by the MIUR project CINA: Compositionality, Interaction, Negotiation, Autonomicity for the future ICT society.

References

- [1] M. Robillard, G. Murphy, Representing concerns in source code, *ACM Trans. Softw. Eng. Methodol.* 16 (1) (2007) 1–38.
- [2] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J.-M. Loingtier, J. Irwin, Aspect-oriented programming, in: 11th European Conference on Object Oriented Programming (ECOOP’97), Lecture Notes in Computer Science, vol. 1241, Springer, Verlag, Helsinki, Finland, 1997, pp. 220–242.
- [3] S. Apel, C. Kästner, An overview of feature-oriented software development, *J. Object Technol.* 8 (5) (2009) 49–84.
- [4] E.W. Wong, S.S. Gokhale, J.R. Horgan, Quantifying the closeness between program components and features, *J. Syst. Softw.* 54 (2) (2000) 87–98.
- [5] P. Greenwood, T. Bartolomei, E. Figueiredo, A. Garcia, N. Cacho, C. Sant’Anna, P. Borba, U. Kulesza, A. Rashid, On the impact of aspectual decompositions on design stability: an empirical study, in: Proceedings of the 21st European Conference on Object-Oriented Programming (ECOOP’07), LNCS, vol. 4609, Springer-Verlag, Berlin, Germany, 2007, pp. 176–200.
- [6] F.C. Filho, N. Cacho, E. Figueiredo, R. Maranhão, A. Garcia, C.M.F. Rubira, Exceptions and aspects: the devil is in the details, in: M. Young, P.T. Devanbu (Eds.), Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE’06), ACM, Portland, OR, USA, 2006, pp. 152–162.
- [7] A. Kaur, K. Johari, Identification of crosscutting concerns: a survey, *Int. J. Eng. Sci. Technol.* 1 (3) (2009) 166–172.
- [8] M. Eaddy, A.V. Aho, G.C. Murphy, Identifying, assigning and quantifying crosscutting concerns, in: Proceedings of 1st International Workshop on Assessment of Contemporary Modularization Techniques (ACoM’07), Minneapolis, USA, 2007.
- [9] E. Figueiredo, C. Sant’Anna, A. Garcia, T.T. Bartolomei, W. Cazzola, A. Marchetto, On the maintainability of aspect-oriented software: a concern-oriented measurement framework, in: C. Tjortjijis, A. Winter (Eds.), Proceedings of the 12th European Conference on Software Maintenance and Reengineering (CSMR 2008), IEEE Press, Athens, Greece, 2008, pp. 183–192.
- [10] E. Arisholm, L.C. Briand, A. Føyen, Dynamic coupling measurement for object-oriented software, *IEEE Trans. Softw. Eng.* 30 (8) (2004) 491–506.
- [11] S. Mouchawrab, L.C. Briand, Y. Labiche, A measurement framework for object-oriented software testability, *J. Inform. Softw. Technol.* 47 (15) (2005) 979–997.
- [12] A. Mitchell, J.F. Power, Toward a definition of run-time object-oriented metrics, in: Proceedings of the 7th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOSSE’03), Darmstadt, Germany, 2003.
- [13] B. Dufour, K. Driesen, L. Hendren, C. Verbrugge, Dynamic metrics for Java, in: Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA’03), ACM Press, Anaheim, California, USA, 2003, pp. 149–168.
- [14] C. Sant’Anna, A. Garcia, C. Chavez, L. Carlos, A. von Staa, On the reuse and maintenance of aspect-oriented software: an assessment framework (SBSE’03), in: Proceedings of the XVII Brazilian Symposium on Software Engineering, Manaus, Brazil, 2003, pp. 19–34.
- [15] R.E. Lopez-Herrejon, S. Apel, Measuring and characterizing crosscutting in aspect-based programs: basic metrics and case studies, in: M.B. Dwyer, A. Lopes (Eds.), Proceedings of the 10th Conference on Fundamental Approaches to Software Engineering (FASE’07), LNCS, vol. 4422, Springer, Braga, Portugal, 2007, pp. 423–437.
- [16] S. Ducasse, T. Girba, A. Kuhn, Distribution map, in: Proceedings of the 22nd IEEE International Conference on Software Maintenance (ICSM’06), IEEE Press, Philadelphia, Pennsylvania, USA, 2006, pp. 203–212.
- [17] E. Figueiredo, N. Cacho, C. Sant’Anna, M. Monteiro, U. Kulesza, A. Garcia, S. Soares, F. Ferrari, S. Khan, F. Filho, F. Dantas, Evolving software product lines with aspects: an empirical study on design stability, in: Proceedings of 30th International Conference on Software Engineering (ICSE’08), ACM, Leipzig, Germany, 2008.
- [18] R.C. Martin, G. Melnik, Tests and requirements, requirements and tests: a Möbius strip, *IEEE Softw.* 25 (1) (2008) 54–59.
- [19] J. Heumann, Generating Test Cases From Use Cases, *The Rational Edge* (2001).
- [20] W. Cazzola, A. Marchetto, AOP-HiddenMetrics: separation, extensibility and adaptability in SW measurement, *J. Object Technol.* 7 (2) (2008) 53–68.
- [21] D.J. Pearce, M. Webster, R. Berry, P.H.J. Kelly, Profiling with AspectJ, *Softw.—Pract. Exp.* 37 (7) (2007) 747–777.
- [22] G. Rothermel, R.H. Untch, C. Chu, M.J. Harrold, Test case prioritization: an empirical study, in: Proceedings of the International Conference on Software Maintenance (ICSM’99), IEEE Computer Society, Oxford, UK, 1999, pp. 179–188.
- [23] D. Batory, J.N. Sarvela, A. Rauschmayer, Scaling step-wise refinement, in: Proceedings of the 25th International Conference on Software Engineering (ICSE’03), IEEE Computer Society, Portland, OR, USA, 2003, pp. 187–197.
- [24] I. Aracic, V. Gasiunas, M. Mezini, K. Ostermann, An overview of CaesarJ, *Trans. Aspect-Orient. Softw. Develop.* 1 (1) (2006) 135–173.
- [25] V. Massol, T. Husted, *jUnit in Action*, Manning Publications Co., 2003.
- [26] A. Marcus, V. Rajlich, J. Buchta, M. Petrenko, A. Sergeev, Static techniques for concept location in object-oriented code, in: Proceedings of the 13th International Workshop on Program Comprehension (IWPC’05), IEEE, St. Louis, Missouri, USA, 2005, pp. 33–Ma42.

- [27] N. Wilde, M.C. Scully, Software reconnaissance: mapping program features to code, *J. Softw. Mainten. Evol.: Res. Pract.* 7 (1) (1995) 49–62.
- [28] M. Eaddy, T. Zimmermann, K. Sherwood, V. Garg, G. Murphy, N. Nagappan, A. Aho, Do crosscutting concerns cause defects?, *IEEE Trans. Softw. Eng.* 34 (4) (2008) 497–515.
- [29] M. Marin, A. Van Deursen, I. Moonen, Identifying crosscutting concerns using fan-in analysis, *ACM Trans. Softw. Eng. Methodol.* 17 (1) (2007).
- [30] K. Erni, C. Lewerentz, Applying design-metrics to object-oriented frameworks, in: *Proceedings of the 3rd IEEE International Software Metrics Symposium (METRICS'96)*, IEEE Computer Society, Berlin, Germany, 1996, pp. 64–74.
- [31] M. Revelle, T. Broadbent, D. Coppit, Understanding concerns in software: insights gained from two case studies, in: *Proceedings of the 13th International Workshop on Program Comprehension (IWPC'05)*, IEEE Computer Society, St. Louis, MO, USA, 2005, pp. 23–32.
- [32] A. Marchetto, A concerns-based metrics suite for web applications, *INFOCOMP J. Comput. Sci.* 4 (3) (2005) 11–22.
- [33] C. Wohlin, P. Runeson, M. Höst, M.C. Ohlsson, B. Regnell, A. Wesslén, *Experimentation in Software Engineering*, Springer, 2012.
- [34] V.R. Basili, L.C. Briand, W.L. Melo, A validation of object-oriented design metrics as quality indicators, *IEEE Trans. Softw. Eng.* 22 (10) (1996) 751–761.
- [35] A. Tahir, S. MacDonell, A systematic mapping study on dynamic software metrics, in: *Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM'12)*, IEEE Press, Riva del Garda, Trento, Italy, 2012.
- [36] S.M. Yacoub, H.H. Ammar, T. Robinson, Dynamic metrics for object oriented designs, in: *Proceedings of the 6th International Software Metrics Symposium (METRICS'99)*, Boca-Raton, FL, USA, 1999, pp. 50–61.
- [37] B. Dufour, C. Goard, L. Hendren, O. de Moor, G. Sittampalam, C. Verbrugge, Measuring the dynamic behaviour of AspectJ programs, in: J. Vlissides (Ed.), *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'04)*, ACM Press, Vancouver, BC, Canada, 2004, pp. 150–169.
- [38] R. Geetika, P. Singh, Empirical investigation into static and dynamic coupling metrics, *ACM SIGSOFT Softw. Eng. Notes* 39 (1) (2014) 1–8.
- [39] V. Gupta, J.K. Chhabra, Dynamic cohesion measures for object-oriented software, *J. Syst. Architect.* 57 (4) (2011) 452–462.
- [40] T. Savage, M. Revelle, D. Poshyanyk, FLAT³: feature location and textual tracing tool, in: J. Kramer, J. Bishop, P. Devanbu, S. Uchitel (Eds.), *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE'10)*, Cape Town, South Africa, 2010, pp. 255–258.
- [41] J. Buckner, J. Buchta, M. Petrenko, V. Rajlich, JRipples: a tool for program comprehension during incremental change, in: J.I. MAletic, J.R. Cordy, H. Gall (Eds.), *Proceedings of the 13th International Workshop on Program Comprehension (IWPC'05)*, IEEE, St. Louis, MO, USA, 2005, pp. 149–152.
- [42] S.H. Kan, *Metric and Models in Software Quality Engineering*, Addison-Wesley, Reading, Ma, USA, 2003.
- [43] J.A. Nelder, R. Wedderburn, Generalized linear models, *J. Roy. Stat. Soc.* 135 (3) (2001) 370–384.
- [44] J. Cohen, *Statistical Power Analysis for the Behavioral Sciences*, Lawrence Erlbaum, 1988.
- [45] R.R. Hocking, The analysis and selection of variables in linear regression, *Biometrics* 32 (1976).
- [46] H. Akaike, A new look at the statistical model identification, *IEEE Trans. Automat. Control* 19 (6) (1974) 716–723.
- [47] D.L. McFadden, Quantitative methods for analyzing travel behaviour of individuals: some recent developments, *Behav. Travel Modell.* (1978) 279–318.
- [48] M. Korte, D. Port, Confidence in software cost estimation results based on MMRE and PRED, in: *Proceedings of the 4th International Workshop on Predictor Models in Software Engineering (PROMISE'08)*, ACM, Leipzig, Germany, 2008, pp. 63–70.
- [49] X. Franch, G. Grau, C. Quer, A framework for the definition of metrics for actor-dependency models, in: *Proceedings of the 12th International Conference on Requirements Engineering (RE'04)*, IEEE Computer Society, Washington, DC, USA, 2004, pp. 348–349.
- [50] P. Ponnuthuramalingam, M. Yamunadevi, An effective analysis of object oriented metrics in software quality, *Int. J. Comput. Technol. Inform. Secur.* 1 (2) (2011) 43–47.
- [51] J.-P. Jacquet, A. Abran, From software metrics to software measurement methods: a process model, in: *Proceedings of the 3rd International Software Engineering Standards Symposium (ISESS'97)*, IEEE Computer Society, Walnut Creek, CA, USA, 1997, pp. 128–135.
- [52] N.R. Brown, R.S. Siegle, Metrics and mappings: a framework for understanding real-world quantitative estimation, *Psychol. Rev.* 100 (3) (1993) 511–534.
- [53] M. Genero, D. Miranda, M. Piattini, Defining metrics for UML statechart diagrams in a methodological way, in: *Proceedings of the 2nd Workshop on Conceptual Modeling Quality (IWCMQ'03)*, LNCS, vol. 2814, Springer, Chicago, IL, USA, 2003, pp. 118–128.
- [54] T. Talbi, B. Meyer, E. Stapf, A metric framework for object-oriented development, in: *Proceedings of the 39th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS'01)*, IEEE Computer Society, Santa Barbara, CA, USA, 2001, pp. 164–172.
- [55] J.C. Taveira, J. Saraiva, F. Castor, S. Soares, A concern-specific metrics collection tool, in: *Proceedings of the OOPSLA Workshop on Assessment of Contemporary Modularization Techniques (ACoM'09)*, Orlando, FL, USA, 2009.
- [56] L.C. Briand, J. Wüst, Modeling development effort in object-oriented systems using design properties, *IEEE Trans. Softw. Eng.* 27 (11) (2001) 963–986.
- [57] A. Kaur, S. Singh, K.S. Kahlon, A metric framework for analysis of quality of object oriented design, *World Acad. Sci. Eng. Technol.* 36 (2009).
- [58] D. Coleman, D. Ash, B. Lowther, P. Oman, Using metrics to evaluate software system maintainability, *IEEE Comput.* 27 (8) (1994) 44–49.
- [59] L.C. Briand, J.W. Daly, J. Wüst, A unified framework for coupling measurement in object-oriented systems, *IEEE Trans. Softw. Eng.* 25 (1) (1999) 91–121.
- [60] A. Seffah, N. Kececi, M. Donyaee, QUIM: a framework for quantifying usability metrics in software quality models, in: *Proceedings of the 2nd Asia-Pacific Conference on Quality of Software (APAQS'01)*, IEEE Computer Society, Hong Kong, China, 2001, pp. 311–318.
- [61] J.M. Conejero Manzano, E. Figueiredo, A. Garcia, J. Hernández, E. Jurado, On the relationship of concern metrics and requirements maintainability, *J. Inform. Softw. Technol.* 54 (2) (2012) 212–238.
- [62] A. Marchetto, A. Trentini, A framework to build quality model for web applications, *Int. Arab J. Inform. Technol.* 4 (2) (2007) 168–176.
- [63] K.A. McKeown, E.G. McGuire, Evaluation of a metrics framework for product and process integrity, *Proceedings of the 33rd Hawaii International Conference on System Sciences (HICSS'00)*, vol. 4, IEEE Computer Society Press, Maui Island, Hawaii, USA, 2000, pp. 4046–4051.
- [64] L.C. Briand, J. Daly, V. Porter, J. Wüst, A comprehensive empirical validation of design measures for object-oriented systems, in: *Proceedings of the 5th International Symposium on Software Metrics (METRICS'98)*, IEEE Computer Society, Bethesda, Maryland, USA, 1998, pp. 246–257.
- [65] D. Soni, R. Shrivastava, M. Kumar, A framework for validation of object-oriented design metrics, *Int. J. Comput. Sci. Inform. Secur.* 6 (3) (2009) 46–52.
- [66] B. Kitchenham, S.L. Pfleeger, N.E. Fenton, Towards a framework for software measurement validation, *IEEE Trans. Softw. Eng.* 21 (12) (1995) 929–943.
- [67] L.C. Briand, S. Morasca, V.R. Basili, Property-based software engineering measurement, *IEEE Trans. Softw. Eng.* 22 (1) (1996) 68–86.