On the Effectiveness of Concern Metrics to Detect Code Smells: An Empirical Study

Juliana Padilha[1], Juliana Pereira[1], Eduardo Figueiredo[1], Jussara Almeida[1], Alessandro Garcia[2], Cláudio Sant'Anna[2]

[1] Computer Science Department, Federal University of Minas Gerais, Belo Horizonte, Brazil
{juliana.padilha,juliana.pereira,figueiredo,jussara}@dcc.ufmg.br

[2] Informatics Department, Pontifical Catholic University of Rio de Janeiro (PUC-RIO), Brazil
{afgarcia,santana}@inf.puc-rio.br

Abstract. Traditional software metrics have been used to evaluate the maintainability of software programs by supporting the identification of code smells. Recently, concern metrics have also been proposed with this purpose. While traditional metrics quantify properties of software modules, concern metrics quantify concern properties, such as scattering and tangling. Despite being increasingly used in empirical studies, there is a lack of empirical knowledge about the effectiveness of concern metrics to detect code smells. This paper reports the results of an empirical study to investigate whether concern metrics can be useful indicators of three code smells, namely Divergent Change, Shotgun Surgery, and God Class. In this study, 54 subjects from two different institutions have analyzed traditional and concern metrics aiming to detect instances of these code smells in two systems. The study results indicate that, in general, concern metrics support developers detecting code smells. In particular, we observed that (i) the time spent in code smell detection is more relevant than the developers' expertise; (ii) concern metrics are clearly useful to detect Divergent Change and God Class; and (iii) the concern metric Number of Concerns per Component is a reliable indicator of Divergent Change.

Keywords: Empirical evaluation, Metrics, Code Smells, Concerns

1 Introduction

The modularization of the driving design concerns is a key factor to achieve maintainable software systems [16, 21]. A concern is any important property or area of interest of a system that we want to treat in a modular way [23]. Business rules, distribution, persistence, and security are examples of typical concerns found in many software systems and that are important, albeit hard, to achieve full modularization. The inadequate separation of concerns degrades design modularity and may lead to maintainability-related design flaws [6, 11]. Detection of these design flaws by programmers is far from trivial and requires effective support.

Software metrics are the key means for assessing the system maintainability [3, 7]. The community of software metrics has traditionally explored quantifiable module properties, such as class coupling, cohesion, and interface size, in order to identify maintainability problems in a software project [3, 8, 19, 20]. More specifically, software measurement is also seen as a pragmatic solution to find symptoms of particular design flaws, such as code

smells [17, 19]. Code smells are symptoms that something may be wrong in the system code [12].

Marinescu [19], for instance, relies on traditional metrics to compose strategies aiming to detect code smells. However, some code smells are often a direct result of poor separation of concerns, and traditional module-driven measurement cannot be tailored to quantify properties of concern modularity. Whereas traditional metrics quantify the properties of modules, the concern metrics quantify properties of concerns, such as scattering and tangling [10]. A growing number of concern metrics have been proposed [5, 6] aiming to quantify key characteristics of concerns' implementation. Indeed, concern metrics have been applied with different purposes and used in several empirical studies. They are used, for instance, to compare aspect-oriented and object-oriented programming techniques [4, 11, 13, 14] and to identify crosscutting concerns that should be refactored [6]. However, we still lack empirical knowledge on the effectiveness of concern metrics to support code smell detection.

To fill this gap, this paper presents an empirical investigation of the effectiveness of concern metrics compared with traditional metrics on the identification of code smells. We report the results of a series of experiments relying on two benchmark applications, named Health Watcher [14] and MobileMedia [11]. This study focuses on a two-dimension analysis comparing the trade-offs on the recall and time efficiency of code smell detection. To analyze the recall, we compare classes identified as suspects of exhibiting a code smell with the reference list of code smells provided by the actual developers in each system. We also assess time efficiency based on the recorded time spent by each subject in the experimental tasks.

This empirical study involved 54 subjects, which were divided into three groups. Subjects of each group participated on the analysis of one of three different sets of metrics: (i) *only traditional metrics*, (ii) *only concern metrics*, and (iii) both traditional and concern metrics, called *hybrid metrics* from now on. These metrics were previously applied to the source code of both target systems. Subjects then analyzed the values of metrics aiming to detect three specific code smells, namely Divergent Change [12], Shotgun Surgery [12], and God Class [22]. Our overall results confirmed that concern metrics, in fact, contribute to improve the detection of code smells. More specifically, this study shows that (i) concern metrics are clearly useful to detect Divergent Change and God Class; (ii) the subject's level of experience does not have significant impact on detection rates; (iii) time explains most of the variations observed in detection rates; and (iv) recall of each metric suite is largely dependent on the adequacy of each metric to quantify a property explicitly mentioned in the smell definition.

The rest of this paper is organized as follows. Section 2 summarizes the concepts of software metrics and code smells. Section 3 describes the study procedures. Sections 4 and 5 reports and discusses the main results of this empirical study. Section 6 highlights the limitations of our study and discusses related work. Section 7 concludes this paper and points out directions for future work.

2 Software Metrics and Code Smells

Software metrics have played an important role in understanding and analyzing software systems [3, 7, 17]. For the purpose of this study, software metrics can be divided into three

sets: traditional metrics (Section 2.1), concern metrics (Section 2.2), and hybrid metrics; i.e., a combination of both traditional and concern metrics. Section 0 describes the three code smells that we investigate in this study.

2.1 Traditional Metrics

We selected a set of the most widely used metrics to be a baseline in this study. The selected set includes object-oriented (OO) metrics proposed by Chidamber and Kemerer [3] and well-documented metrics in the software engineering literature [7]. Table 1 summarizes the metrics used in this study, while detailed definitions can be found elsewhere [3, 7].

Table 1. Definitions of Traditional Metrics.

Metric	Definition
Coupling between Objects (CBO)	Number of classes from which a class calls methods or accesses attributes.
Lack of Cohesion in Methods (LCOM)	It divides pairs of methods that do not access common attributes by pairs that access common attributes.
Lines of Code (LOC)	Total number of lines of code.
Number of Attributes (NOA)	Number of attributes defined in a class.
Number of Methods (NOM)	Number of methods defined in a class.
Weighted Methods per Class (WMC)	Number of methods and their parameters in a class

We selected the most common and widely used traditional metrics for several reasons. First, it is still not well known whether some particular combinations of these metrics can precisely detect specific code smells. Hence, finding combinations involving either concern or traditional metrics might be a relevant result of this paper. Second, we aim to select a reduced number of metrics since many metrics could make the analysis harder and with redundant measurements. Finally, the selected metrics have been used in previous work [8, 11, 19] and they seem to assist developers in software maintenance tasks.

2.2 Concern Metrics

Concern metrics have been defined aiming to capture modularity properties associated with the realization of concerns in software artifacts [10]. Their goal is the identification of specific design flaws [6] or design degeneration caused by poor modularization of concerns [9]. Some recent studies [6, 8] have also shown that concern metrics can be useful indicators of defect-prone modules. Concern is something that you want to treat as a modular unit, including non-functional requirements and programming language idioms [23]. Concern metrics rely on a mapping between concerns and design elements [9, 10]. The mapping consists of assigning a concern to the corresponding design elements that realize it. Table 2 presents a brief definition of the concern metrics evaluated in this paper.

Table 2. Definitions of Concern-based Metrics.

Metric	Definition
Concern Diffusion over Components (CDC)	Number of classes whose main purpose is to contribute to the implementation of a concern and the number of other classes that access them.
Concern Diffusion over Operations (CDO)	Number of methods whose main function is to implement a concern.
Concern Diffusions over LOC (CDLOC)	Number of transition points for each concern through the lines of code. Transition points are points in the code where there is a "concern switch".
Number Concerns per Component (NCC)	Number of concern in each class.

A more detailed description and discussion of these metrics can be found elsewhere [4, 6, 10, 13]. These metrics were selected for evaluation in this paper because they have been successfully used in a number of studies related to software maintainability [11, 13, 14].

However, no systematic study has been performed to evaluate whether these concern metrics support code smell detection.

2.3 Code Smells

Code smells were proposed by Kent Beck in Fowler's book [12] as a mean to diagnose symptoms that may be indicative of something wrong in the system code. It describes a situation where there are hints that suggest a design flaw. This paper investigates the use of concern metrics to detect three code smells, namely Divergent Change [12], Shotgun Surgery [12] and God Class [22], which are described below. These code smells were chosen because they recurrently appear in software systems. Furthermore, previous studies have related these code smells with poor modularization of concerns [2, 19].

Divergent Change. This smell occurs when one class is often changed in different ways for different reasons [12]. For example, we have to change three methods of a class every time we get a new database or we have to change other four methods every time there is a new financial instrument. Depending on the number of assignments of a given class, it may undergo unrelated changes. The fact that a class undergoes various kinds of changes can be associated with a symptom of concern tangling [2].

Shotgun Surgery. This code smell is somehow the opposite of Divergent Change. We identify a Shotgun Surgery instance every time we make a kind of change that leads to a lot of small changes in many different classes [12]. In other words, this code smell can lead to small changes in classes that have a common concern [2].

God Class. This code smell describes an object that knows too much or does too much [22]. It represents a class that has grown beyond all logic to become the class that does almost everything in the system [22]. In a different view, we can say that God Class implements too many concerns and, so, has too many responsibilities [2].

3 Study Settings

This study aims at evaluating the effectiveness of concern metrics in detecting code smells. Our study relies on traditional metrics as baseline. Therefore, we perform a comparative analysis between traditional and concern metrics in order to identify whether the latter supports the former in detecting three specific code smells. Section 3.1 introduces the two target systems. Sections 3.2 and 3.3 present, respectively, the reference list of code smells and background information for the subjects that took part in this study. Finally, Section 0 explains the tasks assigned to each subject.

3.1 Target Systems

Our study involved two software systems: Health Watcher [14] and MobileMedia [11]. These systems were selected because they have been previously used in other maintainability-related studies [4, 8, 11, 18], and we have access to their developers and experts. Therefore, we were able to recover a reference list of actual code smells for each analyzed system (see Section 3.2). A brief description of the Health Watcher and MobileMedia functionalities and their key concerns are described below.

Health Watcher. It is a Web-based information system that supports the registration and management of complaints to the public health system [14]. This system has about 6 KLOC. Some concerns implemented in Health Watcher that we used are: *Business*, *Concurrency*, *Distribution*, *Exception Handling*, *Persistence*, and *View*.

MobileMedia. Our study also involved the 7th version of the MobileMedia system [11]. This system is a software product line (SPL) for applications that manipulate photo, music and video on mobile devices, such as mobile phones. It is an open source project with about 8 KLOC. The concerns of our interest in MobileMedia are: *Sorting*, *Favorites*, *Exception Handling*, *Security*, and *Persistence*.

3.2 Code Smells Reference List

Before conducting the study, we performed a systematic code analysis of Health Watcher and MobileMedia aiming to determine which classes were affected by the relevant code smells. We also relied on two experts in each system to help us building a reference list for each analyzed code smell. These experts participated of the development, maintenance, or assessment of the systems. Our goal was to detect actual instances of each code smell in both systems. Table 3 presents classes in the final reference list of each code smell per system.

Reference List Protocol. Each expert was instructed to individually use their own strategy for detecting code smells in the system classes. As a result, different strategies were used. One expert focused on code inspection following more traditional code analysis. Following a different path, another expert used a complementary set of automated detection strategies [18] to identify candidate instances of the three code smells. For each code smell, the sets of potential instances – one set from each expert – were not exactly the same, although they have many classes in common (approximately 80% and 75% for Health Watcher and MobileMedia, respectively). In order to achieve a consensus, we promoted discussions among experts of the same system. The result of their discussion was recorded as a joint decision and double-checked by ourselves.

Table 3. Code Smell Reference List for MobileMedia.

System	Code Smell	Classes in the Reference List
Health Watcher	Divergent Change	EmployeeRecord, HealthWatcherFacade, HealthUnitRecord, IFacade, PersistenceMechanism, IPersistenceMechanism, ServletInsertEmployee, ServletSearchComplaintData, ServletUpdateComplaintData, ServletUpdateHealthUnitData, ComplaintRecord, HealthWatcherFacadeInit
	Shotgun Surgery	EmployeeRepositoryRDB, IEmployeeRepository, ComplaintRecordRDB, IComplaintRepository, IPersistenceMechanism, PersistenceMechanism, IHealthUnitRepository, HealthUnitRepositoryRDB
	God Class	HealthWatcherFacade, HealthWatcherFacadeInit, PersistenceMechanism
Mobile Media	Divergent Change	ImageMediaAccessor, MediaController, MediaAcessor, MediaListController
	Shotgun Surgery	ControllerInterface, MediaAccessor, ScreenSingleton

3.3 Background of Subjects

This study involved a set of 54 subjects, named S1 to S54, from two different institutions (UFMG/Brazil and Lancaster/UK). Subjects from the first institution were 11 young IT

professional taking an advanced SE course, 4 PhD candidates, and 12 undergraduate students. Subjects from the second institution were 14 PhD candidates and 13 undergraduate students. We organized subjects in such a way that each group worked with only one set of metrics: traditional metrics, concern metrics, or hybrid metrics. The study was performed using the OO designs of both Health Watcher and MobileMedia systems. We conducted 13 rounds of the experiment in different dates. Subjects were organized as follows: (i) 24 subjects detected Divergent Change in 6 rounds, (ii) 20 subjects detected Shotgun Surgery in 5 rounds, and (iii) 10 subjects detected God Class in 2 rounds. Health Watcher was used by subjects of Lancaster to detect all three code smells, while MobileMedia was used by subjects of UFMG to detect Divergent Change and Shotgun Surgery. Further details about the distribution of subjects are available at the project website [1].

Before running the experiment, we used a background questionnaire (also available at [1]) to balance previous knowledge of each subject. Table 4 summarizes knowledge that subjects claimed to have in the background questionnaire. Although the subjects were asked to answer the questionnaire, it was not required to perform the experiment. In fact, we asked subject to indicate their level of knowledge by choosing one of the following options: none, few, moderate, and high experience. Some subjects - annotated in the last column (No Answer) in Table 4 - have not answered the questionnaire. The other columns list subjects who claimed to have knowledge moderate or high in a particular skill.

Table 4. Background Data of Subjects

Divergent Change		***Traditional***	***Concern***	***Hybrid***	***No Answer***
Knowledge	Class Diagram	S5 - S6	S9 - S11	S14 - S24	S1, S2, S3, S7, S8, S12, S13, S18
	Java Programming	S5 - S6	S9 - S11	S14 - S24	
	Measurement	-	S9	S16, S20, S22, S24	
	Academic Experience	S4, S6	S9	S19, S21-S24	
	Work Experience	S5	S10,S11	S14 - S17, S20	
Shotgun Surgery					
Knowledge	Class Diagram	S28, S29	S31, S32	S34 - S37	S25, S26, S30, S33, S38
	Java Programming	S28, S29	S31, S32	S34 - S37	
	Measurement	-	S31	S35, S36	
	Academic Experience	S27, S29	S31	S39, S41-S44	
	Work Experience	S28	S32	S33 - S37, S40	
God Class					
Knowledge	Class Diagram	S46	S48 - S50	S51- S54	-
	Java Programming	S45, S46	S48 - S50	S51- S54	
	Measurement	-	S49 - S50	S52 -S54	
	Academic Experience	S46, S47	S49, S50	S52-S54	
	Work Experience	S45	S48	S51	

Subjects answered questions about their level of knowledge with respect to Class Diagrams, Java Programming, and Software Metrics. Furthermore, they indicated their previous academic and work experience. Some subjects do not appear in a row because they have few or none experience in that particular topic. For instance, with respect to work experience in detecting Divergent Change, subjects S1 to S3 (and others) have not answered the questionnaire, while subjects S4, S27, S39-S44, and S47 claimed to have none or little knowledge in Java Programming. In general, excluding 13 subjects who have not answered the background questionnaire, we have observed that (i) about 60% of the subjects have moderate to high knowledge in Class Diagram and Java Programming; (ii)

71% of the subjects have moderate to high knowledge in at least one topic; and (iii) only 31% of the subjects have low to none knowledge in all topics. Therefore, in general, all subjects have at least basic knowledge required to perform the experimental tasks, and subjects are fairly distributed among the groups of metrics.

3.4 Experimental Tasks

The study was preceded by a 30-minute training session to allow subjects to familiarize themselves with the evaluated metrics and the target code smells. After the training session, each subject received a document containing: (i) a brief explanation and a partial view of the system design as a Class Diagram, and (ii) a description of the concerns involved in the respective analyzed system. The document also described steps and guidelines that subjects should follow, the questions they should answer, and information they should register.

In addition, we provided subjects with the results of the metrics in the respective system under analysis. In order to identify the classes with code smells, we asked subjects to reason about the metrics and identify which of them (alone or combined with other metrics) provide relevant indicators based on the code smell description. We also asked subjects to register the time taken to conclude the experimental tasks and to explain which metrics they used or not to detect each code smell. Each group of subjects (traditional, concern or hybrid) only had access to the results of metrics to which they were assigned. Subjects had no access to the system source code.

4 Results

This section presents the results of our experiments. Section 4.1 introduces the recall and precision metrics, while Sections 4.2 to 4.4 report the results per code smell.

4.1 Evaluation Metrics: Recall and Precision

We rely on three metrics, namely True Positive (TP), False Positive (FP), and False Negative (FN), collected based on the reference lists (Section 3.2). True Positive and False Positive quantify the number of correctly and wrongly identified code smells by a subject. False Negative, on the other hand, quantifies the number of code smells a subject missed out. Based on these metrics, we quantify recall and precision, presented below, to support our analysis. Recall measures the fraction of relevant classes listed by a subject. Relevant classes are classes that appear in the reference list (TP + FN). Precision measures the ratio of correctly detected smells by the total classes a subject listed (TP + FP).

$$\text{Recall (R)} = \frac{TP}{TP + FN} \qquad \text{Precision (P)} = \frac{TP}{TP + FP}$$

We focus our discussion mainly on recall because it is a measure of completeness. That is, high recall means that the subject was able to identify most code smells in the system. High precision, on the other hand, means that a subject indicated more relevant (TP) than irrelevant (FP) code smells. For code smell detection, a large number of false positives are preferred over a large number of false negatives, because manual inspection, which is inevitable, tends to uncover false positives.

4.2 Concern Metrics Support Divergent Change Detection

Table 5 presents the results for the identification of Divergent Change. Rows in this table present three pieces of data: Recall (R), Precision (P), and the Time (T) in minutes used by subjects to complete their tasks. In total, 24 subjects had to identify Divergent Change in the target systems. Table 5 shows that subjects in the concern and hybrid groups achieved better results than those in the traditional group. The average recall of the concern group was 62%. Two out of five subjects in this group identified all code smells (100% of recall). On the other hand, the best achievement by a subject using only traditional metric was 33% of recall. Results of subjects in the hybrid group vary from 0% to 100% of recall (S19 and S16) being on average 41%. These results reveal that, even when analyzed in isolation, concern metrics are an effective means for Divergent Change detection.

4.3 Hard to Detect Shotgun Surgery with Metrics

Table 6, which follows the same structure of Table 5, presents the results for Shotgun Surgery. Note that no group of subjects stands out with good results in this scenario. In fact, only one subject in each group achieved more than 60% of recall: S28 scored 67% analyzing traditional metrics, S30 scored 75% in the concern group, and S35 scored 67% of recall analyzing hybrid metrics. The concern group performed a little better: all subjects scored more than 25% of recall and the average recall was 44%. However, the poor detection rates for almost all subjects suggest that the used metrics cannot properly indicate Shotgun Surgery instances.

Table 5. Results for Divergent Change

Group	**Traditional**						**Concern**						
Subject	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10	S11		
R(%)	17	17	17	33	25	25	100	100	33	25	50		
P(%)	67	50	40	50	17	25	63	100	100	25	29		
T(min)	15	15	40	38	41	36	26	29	29	15	33		
Group	**Hybrid**												
Subject	S12	S13	S14	S15	S16	S17	S18	S19	S20	S21	S22	S23	S24
R(%)	75	8	25	50	100	25	50	0	50	25	50	25	50
P(%)	100	50	75	25	67	33	40	0	67	17	40	17	50
T(min)	40	31	23	36	27	39	24	11	18	19	13	13	12

Table 6. Results for Shotgun Surgery

Group	**Traditional**					**Concern**							
Subject	S25	S26	S27	S28	S29	S30	S31	S32					
R(%)	13	13	0	67	33	75	25	33					
P(%)	25	33	0	25	25	35	40	25					
T(min)	6	10	27	12	14	13	28	14					
Group	**Hybrid**												
Subject	S33	S34	S35	S36	S37	S38	S39	S40	S41	S42	S43	S44	
R(%)	13	50	67	33	33	33	0	0	33	0	0	0	
P(%)	25	80	6	33	25	33	0	0	20	0	0	0	
T(min)	35	14	19	15	4	10	14	9	21	3	7	5	

In addition to a poor recall, almost all subjects (except S34) also had low precision rates. In fact, more than half of the Shotgun Surgery instances detected by the subjects were incorrect, regardless of the metrics used. Interestingly, the subjects detecting Shotgun Surgery in general spent less time (on average) in their tasks than the subjects assigned to detect other code smells. That is, although subjects could not succeed detecting Shotgun Surgery, they did not take much longer to conclude their tasks. This result might indicate

that, if developers do not have appropriate means to detect a code smell, they give up with their duties soon.

4.4 Joint Data Analysis Favor God Class Detection

Table 7 presents the results of God Class. Data in this table suggests that traditional metrics when used in isolation do not offer appropriate means to detect God Class. Two subjects (S45 and S46) in the traditional group scored only 33% of both recall and precision. This low performance is much worse than the one achieved by the concern and hybrid groups. For example, two out of three subjects in the concern group and three out of four subjects in the hybrid group scored 100% of recall. Subjects S49 and S51 are exceptions. In addition, S52 in the hybrid group achieved full precision and recall. Therefore, joint analysis of concern and traditional metrics seems to succeed in detecting this particular code smell.

Table 7. Results for God Class

Group	Traditional			Concern			Hybrid			
Subject	S45	S46	S47	S48	S49	S50	S51	S52	S53	S54
R(%)	33	33	67	100	67	100	33	100	100	100
P(%)	33	33	67	75	100	75	50	100	60	75
T(min)	18	25	27	37	66	43	22	53	51	35

5 Statistical Analysis and Discussions

This section aims to answer three research questions. We focus on the most interesting results, but the complete raw data can be found on the project website [1]. Section 5.1 analyzes the recall of concern metrics compared to the traditional metrics. Section 5.2 discusses to which extent the background of subjects and the time spent impact the recall of code smell detection. Section 5.3 analyzes possible combinations of metrics that increases the recall of identifying each code smell.

5.1 Comparing Concern Metrics and Traditional Metrics

The main goal of this paper is to evaluate the effectiveness of concern metrics to detect code smells. Towards that goal, this section aims to answer the following specific research question: ***RQ1.*** *How accurate do concern metrics perform in comparison with traditional metrics to detect code smells?*

We start by investigating whether the type of system (Health Watcher and MobileMedia) influences the detection of code smells. Table 8 shows average recall results for traditional, concern and hybrid metrics, along with corresponding values of variance, sample size (i.e., number of subjects who participated in the experiment) and 90% confidence intervals. Results are presented separately for each system - Health Watcher and MobileMedia - and for each type of code smell. We also show results for all code smells combined (row *All*). In order to check for statistically significant differences across systems, metrics and/or types of code smell, we perform an unpaired t-test[1] with 90% confidence level [15].

Focusing first on the use of traditional metrics to detect code smells in general (i.e., row *All*); we note that the confidence intervals computed for subjects who used these metrics for the two systems do not overlap. Therefore, we can state that the results for the two systems

[1] We perform an analysis of unpaired observation since we got independent samples from two populations.

are significantly different at the 90% confidence. As shown in Table 8, the results for the system analyzed by a larger number of subjects (37) - MobileMedia – are significantly better (44% higher recall, on average). In other words, detection of code smells using the traditional metrics leads to higher recall while using the MobileMedia system. On the other hand, the two confidence intervals computed for the group of subjects who used concern metrics do overlap. This fact indicated that the results for both systems are *not* statistically different, with 90% confidence. The same behavior is observed for the group of subjects who used hybrid metrics. In other words, whereas the system used does impact the detection of code smells using traditional metrics, the detection using concern and hybrids metrics is not significantly influenced by it.

Table 8. Confidence Intervals (CI)

Systems	**Health Watcher (HW)**			**Mobile Media (MM)**		
Groups	***Traditional (T)***	***Concern (C)***	***Hybrid (H)***	***Traditional (T)***	***Concern (C)***	***Hybrid (H)***
All	(9,1;22,4)	(37,5;95,7)	(10,9;57,5)	(27,8;53,1)	(38,5;86,5)	(27,1;52,7)
DC	(11,6;30,4)	(12,5;142,9)	(-22,7;94,8)	(23,9;26,5)	(-41,4;116,4)	(27,2;57,9)
SS	(-4,0;21,3)	(-107,9;207,9)	(-85,3;148,3)	(-57,3;157,3)	(28,9;37,9)	(6,4;33,4)
GC	-	-	-	(11,2;77,4)	(56,9;121,1)	(43,8;123)

Next, we applied the unpaired t-test (90% confidence level) to evaluate whether the concern metrics lead to significantly different results compared to the other groups of metrics for a fixed system, considering all code smells combined (row *All*). We found that the concern metrics produce significantly higher recall, compared to traditional metrics for the Health Watcher system. For the MobileMedia system, there is a statistical tie, at 90% confidence, though average results are better for the concern metrics. Moreover, we also found that the concern metrics outperform the hybrid metrics in both systems. Thus, we can state that, in general, concern metrics are the best ones, among those analyzed, for the detection of three types of code smells studied. Our intuition is that when the subjects use a greater set of metrics, such as hybrid metrics, they are not likely to obtain better accuracy compared to the concern metrics, since the quantity of metrics could hinder the detection of the code smell. We may argue that concern metrics would be more time efficient because (i) the set only includes four metrics, and (ii) their definitions capture concerns properties that might be related to the code smells.

We also examine whether the type of code smell detected influences the recall of concern metrics in comparison with traditional ones. We restrict our analysis to two code smells, Divergent Change and Shotgun Surgery, because God Class was not analyzed on MobileMedia. Our results indicate that there is no significant difference between the two systems in terms of recall, for any code smell. In other words, subjects were able to recover around the same rates of code smells, regardless of the analyzed system. This is an interesting result because it supports the claims that metrics abstract most of the system complexity [7]. Therefore, metric-based detection of code smells is expected to scale up to larger systems.

After ascertaining that the difference between the systems in terms of recall is not significant, we applied t-tests (90% confidence level) to compare concern metrics against traditional and hybrid metrics for each of three code smells separately, considering the results for both systems together (Table 8). Our results show that the superiority of the concern metrics varied according to the type of code smell. We observed that the use of concern metrics was consistently better in comparison with traditional metrics in the

Divergent Change and God Class detection cases. However, the difference between both types of metrics for Shotgun Surgery is not statistically significant (with 90% confidence). Additionally, we observed that the difference between concern and hybrid metrics is not significant, independently of the type of code smell to be identified. These results indicate that the accuracy of the metric suite is largely dependent on the adequacy of each metric to quantify a property explicitly mentioned in the smell definition. For instance, God Class is characterized by the "high amount of class members with the realization of multiple responsibilities" [12].

This property is better captured by concern metrics. This probably explains why the concern metrics outperformed the traditional ones for God Class detection. Data also suggests that detecting Divergent Change with only traditional metrics seems harder when compared to the support of concern metrics. The explanation could be that this code smell is closely related to poor separation of concerns. Divergent Change often occurs when several concerns are tangled into a module [2]. Therefore, this module is likely to be changed by different reasons. Focusing on subjects that used concern metrics (concern and hybrid groups), it is interesting to note that 10 out of 18 subjects in either groups achieved 68% of recall on average.

5.2 Background of Subjects

Our goal in this section is to analyze whether the background of subjects can impact the results. In other words, we aim to answer the following research question: ***RQ2.*** *Does background of subjects impact the efficiency of the detected code smell?*

To answer RQ2, we evaluate the impact of both the background of subjects and the time spent by them on the effectiveness of the detection when using concern metrics. To that end, we apply a 2^k full factorial design with k=2 factors, namely the developers' work experience and the time spent in detected code smells [15]. As discussed in Section 3.3, all subjects have at least basic knowledge in the relevant topics of software development, namely UML Class Diagram, Java Programming, and Measurement. Therefore, we decided to draw this analysis with respect to work experience of subjects which varied a lot among subjects [1]. In this analysis, we excluded subjects that did not answer the background questionnaire (Section 3.3).

We focus on the recall of the detected code smells using the concern metrics as the response variable. For this analysis, we consider the results for all code smells and both systems together. Since we did not observe statistical difference in the recall of detection across systems when using concern metrics (Section 5.1), we grouped the results for both systems together for this analysis. Moreover, we also consider all three code smells indistinctly.

We divided subjects into two categories according to their work experience: (i) no experience indicates those subjects who never worked, or worked for fewer than 6 months, and (ii) some experience identifies those subjects who worked for at least 6 months in software development industry. Additionally, we also divided subjects into two categories according to the time spent in detected code smells: (i) short time indicates those subjects who took less than 33 minutes (overall average) to detect the code smells, and (ii) long time indicates those subjects who took at least 33 minutes.

In general, results show that the recall tends to increase with the work experience and the time spent in the detection, as one might expect. In order to quantify the relative impact of

each of these factors on the subjects' recall, we compute the percentage of the variation in the measured recall that can be credited to each factor in isolation, as well as to the interaction of both factors. The higher the percentage of variation explained by a factor/interaction, the more important it is to the response variable [15].

Out of the total variation observed in our measurements, 96% can be attributed to the time spent in the detection, whereas only 4% is due to variations in the subjects' work experience and 1% can be attributed to the interaction of these two factors. Thus, both the work experience factor and the interaction seem of little importance to the final recall, compared to the time subjects spent in detecting the code smells. The latter has a major impact on the final recall. Indeed, the results clearly show that the subjects who spent more time to analyze the concern metrics achieved the better results in terms of recall. One possible explanation is the complexity of concern metrics, which require more time from subjects to successfully perform the detection. Additionally, even the subjects who have no experience tend to obtain a higher recall when they spend a longer time to detect the smell.

5.3 Metrics Flocking Together

In this section, we analyze possible metrics that might be useful to detect specific code smells and answer the following research question. ***RQ3.*** *Is there a combination of metrics that increases recall of code smell detection?*

As explained in Section 3.4, subjects reported the metrics they considered useful for each code smell. Based on their answers, we analyzed in this section the metrics that were considered useful by at least three subjects. In order to determine which metrics were used together to detect code smells, we performed analysis of subjects who used the same metrics and scored high in terms of recall. Table 9 shows the metrics that at least three subjects claimed to have used for Divergent Change. In this case, we also restricted our analyzes to metrics with average of recall higher than 30%. Both the Number Concern per Component (NCC) and Lack of Cohesion in Methods (LCOM) metrics were considered useful to detect Divergent Change by eleven subjects. Subjects that considered these metrics useful achieved 60% and 34% of recall in average, respectively. Additionally, the concern metric Concern Diffusion over Components (CDC) was considered useful by 3 subjects. It is interesting to observe that subjects that considered concern metrics NCC and CDC useful achieved better results in terms of recall.

Table 9. Metrics Considered Useful for Divergent Change

Metrics	NCC	LCOM	CDC	LOC
Subjects who used this metric	S7, S8, S9, S11, S12, S14, S15, S16, S20, S23, S24	S1, S2, S4, S6, S12, S13, S14, S15, S17, S22, S24	S8, S10, S23	S2, S17, S20
Average of recall	60%	34%	50%	31%

In particular, NCC seems the most effective metric (among the analyzed ones) to detect Divergent Change. For instance, S7, S8, S12 and S16 used NCC - solo or in combination with other metrics - and achieved 94% of recall. We also observed that subjects who indicated NCC as not being useful achieved less than 11% of recall; as it is the case of S10, S13 and S19. Additionally, subjects who indicated NCC and LCOM as being useful achieved 50% of recall in average. For instance, we observed that the metrics were used together by subjects S12 and S15. These subjects achieved 75% and 50% of recall respectively. Interestingly, while S12 had 100% of precision, S15 had only 25%. We also

observed that subjects who indicated NCC and LCOM as not being useful achieved 0% of recall; as it is the case of S19.

Since most subjects had poor performance for detecting Shotgun Surgery, Table 10 presents metrics considered useful by at least three subjects when detecting this code smell. Coupling between Object (CBO) was considered useful by eleven subjects. However, these subjects achieved only 15% of recall in average. On the other hand, five subjects indicated Concern Diffusion over Components (CDC) as being useful and achieved 23% of recall in average. In addition, Number Concern per Component (NCC) was considered useful by four subjects who achieved 42% of recall in average. Hence, the concern metrics NCC achieved better results in terms of recall. A combination of these metrics, i.e.., NCC and CBO, was used together by subject S37 who achieved 33% of recall. In fact, all subjects that used NCC, solo or in combination with other metrics, scored higher than 30% of recall. This is the case of subjects S32 (33%), S35 (67%), S36 (33%), and S37 (33%). However, a combined analysis of Tables 6 and 10 does not allow us to conclude that these metrics (and any other) are good to detect Shotgun Surgery due to the global symptoms associated with this code smell.

Table 10. Metrics Considered Useful for Shotgun Surgery

Metrics	CBO	CDC	NCC
Subjects who used this metric	S25-S29, S37, S39, S40, S42-S44	S30, S31, S33, S40, S43	S32, S35, S36, S37
Average of Recall	15%	23%	42%

Table 11 shows for God Class the metrics (i) considered useful by at least three subjects and (ii) with average of recall for these subjects higher than 60%. Coupling between Object (CBO) and Lack of Cohesion in Methods (LCOM) were considered useful to detect God Class by at least four subjects. Subjects using these metrics achieved about 67% of recall in average. On the other hand, three metrics also considered useful achieved recall rates above 85%, namely Weighted Methods per Class (WMC), Lines of Code (LOC), and Concern Diffusions over LOC (CDLOC). This result suggests that size metrics, such as LOC and WMC, and the concern metric CDLOC are good indicators of God Class. Additionally, we observed some cases of metrics that were used together. WMC with LOC seems the best combination of metrics. It was used by S52 and S53 and worked well since both subjects achieved 100% of recall. In addition, the combination of WMC and CBO, was used by S47 and S53 and worked well since subjects achieved 67% and 100% of recall respectively. Another case was the combination of CBO with LCOM used by the subjects S51, S53 and S54. These Subjects achieved 78% of recall in average.

Table 11. Metrics Considered Useful for God Class

Metrics	CBO	LCOM	WMC	LOC	CDLOC
Subjects who used	S46, S47, S51, S54	S45, S51, S53, S54	S47, S52, S53	S52, S53, S54	S48, S49, S53
Average of recall	67%	67%	89%	100%	89%

6 Threats to Validity and Related Work

The conclusions obtained here are restricted to the involved metrics, code smells, and target software systems. These limitations are typical of studies like ours. Although we acknowledge these limitations, we note that our study fills a gap in the literature by reporting original analyses on the benefits of using concern metrics for detecting code

smells. Additionally, this paper describes an experimental framework that can be used in further rounds of this study.

Ultimately, the recall of concern metrics depends on how accurate the mapping (assignment) of each concern to code elements was. Fortunately, we observed in a previous study [9] that, apart from Concern Diffusion over Lines of Code (CDLOC), the mapping process does not significantly impact the concern metrics assessed in this paper. Additionally, in order to mitigate this threat, we relied on concern mappings produced by the original developers. Whether the concern mapping was fully correct or not, it just reflects how concern metrics would be used in practice.

Detection strategies of code smells have been the subject of recent studies reported in the literature. They are usually based on exploiting information that is extracted from the source code [6, 8, 9, 11, 14, 17, 19] and rely on the combination of metrics. Metrics has been historically used to detect code smells [17, 19]. Marinescu [19] proposed the use of strategies composed of traditional metrics for detecting code smells. He observed that multiple metrics are required to capture all factors in the code smell definition. He relied on several traditional metrics also used in their study, but have not used concern metrics.

Several studies have used traditional and concern metrics to assess diverse maintainability attributes of software systems, such as instability [11, 14] and error-proneness [6, 8]. Some of these studies [11, 14] rely on concern metrics to support the comparison of aspect-oriented and object-oriented decompositions. Unlike our work, these studies implicitly assume that concern metrics are reliable indicators of the respective quality attribute assessed. This paper, on the other hand, aims to verify whether concern metrics can provide appropriate means to detect code smells.

Eaddy and his colleagues [6] have carried out three experiments to evaluate the usefulness of concern metrics to identify error-prone modules. Their experiments evaluated six concern metrics; two of them are also used in our experiment, namely CDC and CDO. They found a moderate to strong correlation between the concern metrics and defects in modules for all three experiments. The purpose of our study is different, due the fact that we are not focused on error-proneness analysis. Our work complements and extends Eaddy's findings since we observed that concern metrics could also serve as reliable indicators of code smells.

7 Conclusions and Future Work

The evaluation of software maintainability is largely dependent on the availability of metrics that accurately detect code smells. Concern metrics are increasingly being used in empirical studies [4, 11, 13, 14]. Our study aims at examining the effectiveness of concern metrics to detect code smells. Our results revealed that concern metrics are clearly useful to detect Divergent Change and God Class and that experience of developers does not have influence on the effectiveness of code smell detection. Additionally, we observed that the effectiveness of each metric suite is largely dependent on the adequacy of each metric to quantify a property explicitly mentioned in the smell definition. For instance, we observed that the concern metric Number of Concerns per Component (NCC) was efficient to detect Divergent Change even when used alone because it seems to quantify a dimension of module cohesion that is not captured by other metrics.

This study represents a first stepping-stone towards the evaluation of concern metrics to detect code smells. We are currently working on strategies to detect code smells based on the concern metrics we found useful. We also plan to perform further empirical studies to analyze the role of concern metrics at different levels of abstraction, such as architectural and detailed design.

References

1. Data of the Experiment with Metrics: http://www.dcc.ufmg.br/~juliana.padilha/caise2014
2. Carneiro, G. F. et al.: Identifying Code Smells with Multiple Concern Views, Proc. of the Brazilian Symposium on Software Engineering (SBES), 128-137 (2010)
3. Chidamber, S. R. and Kemerer, C. F.: A Metrics Suite for Object Oriented Design. Trans. on Software Engineering (1994)
4. Conejero, J. M. et al.: On the Relationship of Concern Metrics and Requirements Maintainability, Inf. and Sof. Technology (IST) (2011)
5. Ducasse, S. ,Girba, T. and Kuhn, A.: Distribution Map, Proc. of ICSM, 203-212 (2006)
6. Eaddy, M. et al. Do Crosscuting Concerns Cause Defects? IEEE Trans. on Software Engineering, 497-515 (2008)
7. Fenton, N. E. and Pfleeger, S. L.: Software Metrics: A Rigorous and Practical Approach. Thomson (1996)
8. Ferrari, F. et al.: An Exploratory Study of Fault-Proneness in Evolving Aspect-Oriented Programs. Proc. of the Int'l Conf. on Software Engineering (ICSE), 65-74 (2010)
9. Figueiredo, E. et al.: On the Impact of Crosscutting Concern Projection on Code Measurement, Proc. of the Int'l Conf. on Aspect-Oriented Soft. Develop. (AOSD) (2011)
10. Figueiredo, E. et al.: On the Maintainability of Aspect-Oriented Software: A Concern-Oriented Measurement Framework, Proc. of CSMR (2008)
11. Figueiredo, E. et al.: Evolving Software Product Lines with Aspects: an Empirical Study on Design Stability, Proc. of the Int. Conf. on Soft. Engineering (ICSE),261-270 (2008)
12. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison Wesley (1999)
13. Garcia, A.: Modularizing Design Patterns with Aspects: A Quantitative Study. Proc. of the Int. Conf. Aspect Oriented Software Development (AOSD), March 14-18 (2005)
14. Greenwood, P. et al.: On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study, Proc. of ECOOP (2007)
15. Jain, R.: The Art of Computer System Performance Analysis: Techniques for Experimental Design, Measurement, Simulation and Modeling. John Wiley & Sons, pages 1-702 (1991)
16. Kiczales, G. et al.: Aspect-Oriented Programming. Proc. of ECOOP, 220-242 (1997)
17. Lanza, M. and Marinescu, R.: Object-Oriented Metrics in Practice. Springer Verlag (2006)
18. Macia, I. et al.: Are Automatically-Detected Code Anomalies Relevant to Architectural Modularity? Proc. of Int'l Conf. on Aspect-oriented Soft. Dev. (AOSD), 167-178 (2012)
19. Marinescu, R.: Detection Strategies: Metrics-Based Rules for Detecting Design Flaws. Proc. of Int'l Conf. on Software Maintenance (ICSM), pp. 350-359 (2004)
20. Nguyen, T., Nguyen, H., Nguyen, H. and Nguyen, T.: Aspect recommendation for evolving software. Proc. of the Int'l Conf. on Soft. Eng. (ICSE), pp. 361-370,(2011)
21. Parnas, D. L.: On The Criteria to Be Used in Decomposing Systems into Modules. Comm. of the ACM, 15(12), 1053-1058 (1972)
22. Riel, A. J.: Object-Oriented Design Heuristics. Addison-Wesley Professional (1996)
23. Robillard, M. and Murphy, G. Representing Concerns in Source Code, Trans. on Soft. Eng. and Meth. (2007)