Applying and Evaluating Concern-Sensitive Design Heuristics

Eduardo Figueiredo[1], Claudio Sant'Anna[2], Alessandro Garcia[3], Carlos Lucena[3]

[1]Computer Science Department, Federal University of Minas Gerais – UFMG, Brazil
[2]Software Engineering Lab (LES), Computer Science Department, Federal University of Bahia – UFBA, Brazil
[3]Opus Research Group, Computer Science Department, Pontifical Catholic University of Rio de Janeiro – PUC-Rio, Brazil

figueiredo@dcc.ufmg.br, santanna@dcc.ufba.br, {afgarcia, lucena}@inf.puc-rio.br

Abstract

Manifestation of crosscutting concerns in software systems is often an indicative of design modularity flaws and further design instabilities as those systems evolve. Without proper design evaluation mechanisms, the identification of harmful crosscutting concerns can become counter-productive and impractical. Nowadays, metrics and heuristics are the basic mechanisms to support their identification and classification either in object-oriented or aspect-oriented programs. However, conventional mechanisms have a number of limitations to support an effective identification and classification of crosscutting concerns in a software system. In this paper, we claim that those limitations are mostly caused by the fact that existing metrics and heuristics are not sensitive to primitive concern properties, such as either their degree of tangling and scattering or their specific structural shapes. This means that modularity assessment is rooted only at conventional attributes of modules, such as module cohesion, coupling and size. This paper proposes and systematically evaluates a representative suite of concern-sensitive heuristic rules. The proposed heuristics are supported by a tool. The accuracy of the heuristics is assessed through their application to seven systems. The results of analysis indicate that the heuristics offer support for: (i) addressing the shortcomings of conventional metrics-based assessments, (ii) reducing the manifestation of false positives and false negatives in modularity assessment, (iii) detecting sources of design instability, and (iv) finding the presence of design modularity flaws in both object-oriented and aspect-oriented programs.

Keywords

Design heuristics, modularity, crosscutting concerns, software metrics, aspect-oriented software development.

1. Introduction

Software systems are always changing to address new stakeholders' concerns. Design modularity improves the stability of software by decoupling design concerns that are likely to change so that they can be maintained independently. A concern is any consideration that can impact on the design and maintenance of program modules [21]. According to this general definition, concerns can be, for instance, non-functional requirements, domain-specific features, design patterns, or implementation idioms [15, 16]. Despite the efforts of modern programming languages, some concerns cannot be well modularised in the system design and implementation [15]. These concerns, called crosscutting concerns, are often blamed to hinder design modularity and stability [8, 14]. Inaccurate concern modularisations can lead to a wide range of design flaws [9, 14], ranging from concern tangling and scattering [7, 16] to specific code smells, such as feature envies or god classes [30].

Without proper design evaluation mechanisms, the identification of harmful categories of crosscutting concerns can become counter-productive and impractical. Nowadays, metrics and heuristics are the basic mechanisms to support their identification and classification [7]. In fact, metrics and heuristics are traditionally the fundamental mechanisms for assessing design modularity [4, 17, 18, 20]. To date, design modularity assessment either in object-oriented or aspect-oriented designs has been mostly rooted at extensions of module-level metrics [4, 22, 23] that have been historically explored in the software engineering literature. For instance, Sant'Anna [22], Ceccato [3] and their colleagues defined metrics for aspectual coupling and cohesion based on Chidamber and Kemerer's metrics [4]. However, these measures do not treat concerns as first-class assessment abstractions. Similarly, existing heuristic rules [17, 18] are also based on such traditional modularity metrics and do not promote concern-sensitive design evaluation either.

The recognition that concern identification and analysis are important through software design activities is not new. In fact, with the emergence of aspect-oriented software development (AOSD) [16], there is a growing body of relevant work in the software engineering literature focusing either on concern representation and identification techniques [5, 7, 21] or on concern analysis tools [21]. However, there is not much knowledge on the efficacy of concern-driven assessment mechanisms for design modularity. Even though we can qualify some recently-proposed metrics as "concern-oriented" [5, 22], there is a lack of design heuristics to support concern-sensitive assessment. More fundamentally, there is no systematic study that investigates if this category of heuristic rules addresses limitations of conventional metrics and enhances both the processes of classifying crosscutting concerns, and evaluating design modularity and stability.

In this context, the contributions of this paper are fourfold. First, after revisiting existing assessment mechanisms, it discusses the limitations of conventional metrics-based heuristics (Section 2). Second, it presents a suite of heuristic rules with the distinguishing characteristic of exploiting concerns as explicit abstractions in the design assessment process (Section 4). The heuristic rules rely on a set of concern-driven metrics (Section 3) and target at addressing complementary concern-driven design analyses:

1. **Identification and classification of crosscutting concerns** according to their primitive properties, such as tangling and scattering. These heuristics are fundamental as concern scattering tends to both destabilise software modularity properties and facilitate the propagation of changes according to previous empirical assessments [8, 14].

2. **Identification and classification of specific crosscutting patterns** according to more sophisticated properties of crosscutting concerns in artefacts. These heuristics are complementary to the first group of heuristics as it has recently found that it is probably the case that certain recurring forms of crosscutting concerns are not harmful to design stability [7].

3. **Detection of classical bad smells** [10, 20] that are often sensitive to the way concern realisations are organised in the source code, such as feature envies and god classes.

Third, this paper also presents a prototype tool, called ConcernMorph (Section 5), which supports the proposed heuristics. Finally, it provides a systematic evaluation on the accuracy of the concern-sensitive heuristics in the context of seven applications. We have analysed both OO and aspectual designs of such systems, which encompass heterogeneous forms of crosscutting and non-crosscutting concerns (Section 6). The overall results of our evaluation indicate that concern-sensitive design heuristics (Section 7): (i) address most of the shortcomings of conventional assessment mechanisms, (ii) support effective identification and classification of crosscutting concerns with around 80% of accuracy, and (iii) present superior identification rates of crosscutting anomalies (compared to conventional heuristics) that are harmful to modularity and stability both in object-oriented and aspect-oriented programs.

2. Design Modularity Assessment

This section discusses aspect-oriented metrics (Section 2.1) and conventional heuristics (Section 2.2) for modularity evaluation of object-oriented and aspect-oriented (AO) systems. It also points out limitations of these conventional assessment mechanisms (Section 2.2) for modularity assessment. Studies on aspect-oriented measurement and conventional heuristics represent the work related to of the proposed concern-sensitive heuristics. To the best of our knowledge, no other work on concern-sensitive heuristics has been carried out.

2.1. Metrics for Aspect-Oriented Designs

A number of metrics [3, 22, 23] have been recently defined to quantify design modularity properties of AO systems. However, most of these metrics, named AO metrics for short, are based on extensions of metrics for OO design assessment [33], such as Chidamber and Kemerer's Coupling between Objects (CBO) [4]. Table 1 presents four conventional AO metrics used later in this paper (Section 4). Table 1 also includes references (1st column) and a short description of each metric (2nd column). This table includes metrics to quantify the number of components (NC), attributes (NOA), and operations (NOO) of a software design. A component is a unified abstraction to both aspectual and non-aspectual modules. For instance, a component represents either a class or an interface in OO designs, and a component is a class, an interface, or an aspect in AO designs [16]. Similarly, operations can be methods, constructors, advice, or inter-type methods in AO designs.

Table 1. Examples of traditional software metrics [3, 4, 22]

Metrics	Definitions
Number of Components (NC) [22]	Counts the number of classes, interfaces and aspects.
Number of Attributes (NOA) [22]	Counts the number of attributes of each class, interface or aspect.
Number of Operations (NOO) [22]	Counts the number of methods, constructors, and advice of each class, interfaces, or aspect.
Coupling between Components (CBC) [3, 4, 22]	Counts the number of other classes, interfaces, and aspects which a component is coupled to.

There are also some AO metrics based on dependency graphs [23] which focus on specific mechanisms available in AO languages. For instance, Zhao [23] and Ceccato [3] propose suites of metrics that measure coupling and cohesion based on mechanisms and abstractions available in AspectJ [25], such as pointcut, advice, and intertype declarations. We focus on more general AO metrics, such as those one presented in Table 1, in order to allow our approach being applied to a variety of AO languages, such as AspectJ and CaesarJ [26]. Note that, all these AO metrics are rooted at attributes of modules, such as syntax-based module cohesion, coupling between modules, and module interface complexity.

2.2. Conventional Heuristic Assessment

Despite the extensive use of metrics, if used in isolation metrics are often too fine grained to comprehensively quantify the investigated modularity flaw [18, 32]. In order to overcome this limitation of metrics, some researchers [17, 18, 24, 32] proposed a mechanism called *design heuristic rule* (or detection strategy) for formulating metrics-based rules that capture deviations from good design principles. A heuristic rule is a composed logical condition, based on metrics, which detects design fragments with specific problems.

For instance, we present below a heuristic rule proposed by Marinescu [32]. This rule aims at detecting a specific kind of modularity flaw, namely the Shotgun Surgery bad smell [10]. Bad Smells are proposed by Kent Beck in Fowler's book [10] to diagnose symptoms that may be indicative of something wrong in the design. Shotgun Surgery occurs when a change in a characteristic (or concern) of the system implies many changes to a lot of different places [10]. The reason for choosing Shotgun Surgery as illustrative is because it is believed to be a symptom of design flaws caused by poor modularisation of concerns [19]. Therefore, it might be avoided with the use of aspects.

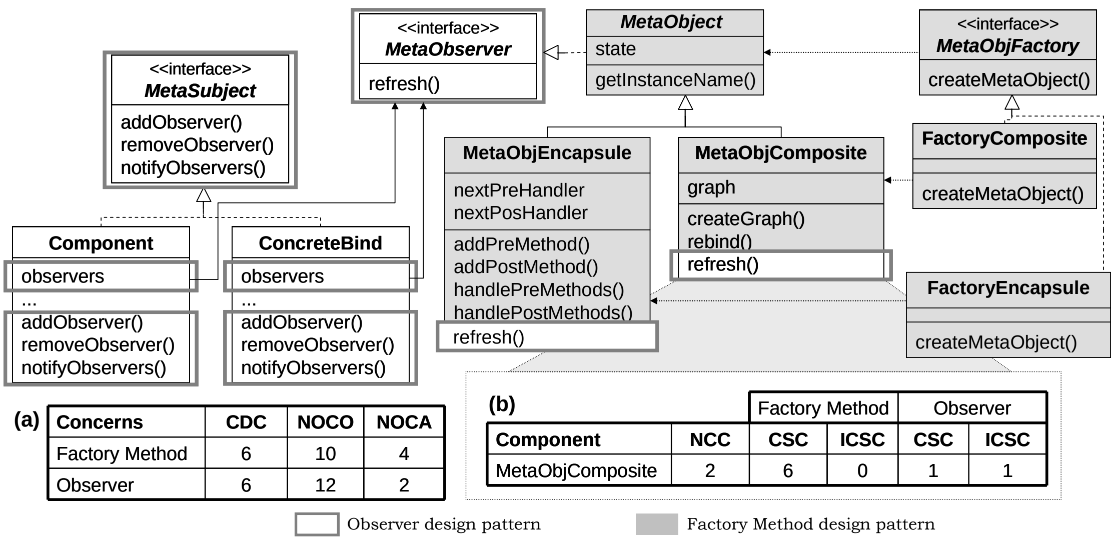
Shotgun Surgery := ((CM, TopValues(20%)) and (CM, HigherThan(10))) and (CC, HigherThan(5))

The Marinescu's heuristic rule for detecting Shotgun Surgery is based on two conventional coupling metrics, namely CM and CC. CM stands for the Changing Method metric [17, 32], which counts the number of distinct methods that access an attribute or call a method of the given class. CC stands for the Changing Classes metric [17, 32], which counts the number of classes that access an attribute or call a method of the given class. *TopValues* and *HigherThan* are filtering mechanisms which can be parameterised with a value representing the threshold. For instance, the Shotgun Surgery heuristic above says that a class should not have CC higher than 5 and should not have methods with the top 20% highest CM if these methods have CM higher than 10. To the best of our knowledge, all current design heuristic rules, such as the one presented above, are based on conventional module-driven metrics. Therefore, a common characteristic of all those heuristics is that they are restricted to properties of modularity units explicitly defined in AO or OO languages, such as classes, aspects, and operations.

2.3. Limitation of Conventional Heuristics

Although many design modularity flaws are related to the inadequate modularisation of concerns [2, 12, 14], most of the current quantitative assessment approaches, such as AO metrics (Section 2.1) and heuristic rules (Section 2.2), do not explicitly consider concern as a measurement abstraction. This imposes certain shortcomings in order to effectively detect and correct design impairments. This limitation becomes more apparent in the age of AOSD since different forms of design composition and decompositions have been brought to separate crosscutting concerns. To illustrate the limitations of conventional metrics-based heuristic rules, we analyse the effectiveness of the Marinescu's rule [18] presented in Section 2.2. This heuristic rule is applied to a partial design showed in Figure 1 with the purpose of detecting the Shotgun Surgery bad smell. This figure shows a partial class diagram realising both Factory Method and Observer patterns [11]. Elements of the design are shadowed to indicate which design pattern they implement.

Applying CC and CM, we obtain CC = 0 and CM = 15 for the MetaSubject interface (Figure 1). Based on these values and computing the Marinescu's heuristic, this interface is not regarded as a suspect of Shotgun Surgery. This occurs because CC is 0, since no class in the system directly accesses MetaSubject. Nevertheless, this interface can be clearly considered as Shotgun Surgery because changes on its methods would trigger many other changes in every class implementing it and potentially in classes calling its overridden methods. For instance, renaming the addObserver() method in the MetaSubject interface causes updates to the classes Component and ConcreteBind (Figure 1) and several other classes which call addObserver().



(a)

Concerns	CDC	NOCO	NOCA
Factory Method	6	10	4
Observer	6	12	2

(b)

		Factory Method		Observer	
Component	NCC	CSC	ICSC	CSC	ICSC
MetaObjComposite	2	6	0	1	1

Figure 1. Concern metrics applied to the Observer and Factory Method patterns

This example aims at showing how conventional heuristic rules are limited to point out the overall influence of one concern – the Observer design pattern in this case – in other parts of the design. Particularly, the Marinescu's rule was not able to detect that a significant number of classes include design elements related to the Observer pattern and, as a consequence, that they could be affected due to a change in this concern. In fact, this rule could not highlight the complete impact of the Observer pattern because it considers only measures based on the module and method abstraction.

3. Concern-Driven Metrics

This section defines concern metrics used by the suite of heuristics presented in Section 4. All metrics defined in this section have a common underlying characteristic that distinguishes them from conventional metrics (Section 2.1): they capture information about concerns traversing one or more design modularity units. A concern is often not defined by the "boundaries" of modules in modelling or programming languages [21], such as components (e.g., classes or aspects), and operations (e.g., methods or advice). Instead, a concern spread over the design elements is realised by a heterogeneous set of these elements.

Metrics Definition. Each concern-driven metric of this section is presented in terms of a definition, the measurement purpose, and an example. As far as the example is concerned, we rely on an OO design slice of a middleware system [2] presented in Figure 1. The concern metrics are computed based on the mapping of concerns to design elements. We used ConcernMapper [27] for supporting semi-automatic concerns mapping and ConcernMorph (Section 5) for concern measurement. The choice of the concerns to be measured depends on the nature of the assessment goals; some examples will be given in this section and through our evaluation (Section 5).

3.1. Concern Scattering and Tangling

The metric *Concern Diffusion over Components (CDC)* [22] counts the number of classes and aspects that have some influence of a certain concern, thereby enabling the designer to assess the degree of concern scattering. For instance, Figure 1 shows that there is behaviour related to the Factory Method pattern in six components (MetaObject, MetaObjFactory, and respective subclasses). Therefore, the value of the CDC metric for the Factory Method concern is six (Figure 1, Table (a)). Unlike CDC, the metric *Number of Concerns per Component (NCC)* quantifies the concern tangling from the system components' point of view. It counts the number of concerns each class or aspect implements. The goal is to support designers on the observance of intra-component tangling degree. The value of this metric for MetaObjComposite is two (Figure 1, Table (b)), since this component implements the concerns of both Factory Method and Observer patterns.

3.2. Concern Materialisation and Coupling

The metric *Number of Concern Attributes (NOCA)* counts the attributes (including inter-type attributes in aspects) that contribute to the realisation of a certain concern. Similarly, *Number of Concern Operations (NOCO)* counts the methods, constructors, and advices that participate in the concern realisation. The goal of NOCA and NOCO is to quantify how many internal component members are necessary for the materialisation of a specific concern. The example in Figure 1 shows that the value of NOCO is ten for the Factory Method design pattern, since this is the number of operations implementing it. In the same example we see that four attributes realise this design pattern (NOCA=4).

Concern-Sensitive Coupling (CSC) quantifies the number of components a given class or aspect realising the concern is coupled to. In other words, CSC counts the number of coupling connections is associated to the concern of interest in a specific component. Similarly, the metric *Intra-Component Concern Sensitive Coupling (ICSC)* counts the number of internal attributes accessed and internal methods called by a concern in a given component. Additionally, these attributes and methods (i) have to implement another concern rather than the assessed one and (ii) cannot be inherited nor introduced in the class by inter-type declaration. Figure 1 shows that Factory Method is coupled to six components in the MetaObjComposite class (CSC=6) and uses no internal member of this class realising the Observer pattern (ICSC=0). Sequence diagrams are required to count CSC and ICSC at the design level, but we omitted them for the sake of simplicity.

4. Concern-Sensitive Heuristics

This section presents heuristic rules to address the limitations of conventional heuristic rules discussed in Section 2. Our main goal is to define heuristics that make the evaluation process sensitive to the design concerns. The proposed heuristics, called concern-sensitive heuristic rules, are defined in terms of combined information collected from concern metrics (Section 3) and conventional modularity metrics. Each heuristic expression embodies modularity knowledge about the realisation of a concern in the respective OO or AO designs. The motivation of concern-sensitive heuristics is to minimise the shortcomings of conventional metrics-based heuristics illustrated in Section 2. Our hypothesis to be tested in Section 5.2 is that most of these problems can be ameliorated through the application of concern-aware heuristics.

Heuristics Structure. All heuristic rules are expressed using conditional statements in the form: *IF <condition> THEN <consequence>*. The condition part encompasses one or more metrics' outcomes related to the design concern under analysis. If the condition is not satisfied, then the concern analysis is concluded and the concern classification is not refined. In case the condition holds, the role of the consequence part is to describe a change or refinement of the target concern classification. The heuristic rules were structured in such a way that the classification is systematically refined into a more specialised category. In other words, when used to find bad symptoms they gradually generate warnings with higher gravity. The generated warnings encompass information that helps the designers to concentrate on certain concerns or parts of the design which are potentially problematic. The proposed heuristics suite is structured to detect three major groups of concern-related flaws: (i) concern diffusion (Section 4.1), (ii) patterns of crosscutting concerns (Section 4.2), and (iii) specific design flaws (Section 4.3).

4.1. Concern Diffusion

Heuristic rules of this section, named concern diffusion heuristics, classify the way each concern manifests itself through the software modularity units. Concern can be classified into one (or more) of the six categories: Isolated, Tangled, Little Scattered, Highly Scattered, Well Modularised, and Crosscutting. A *tangled concern* is interleaved with other concerns in at least one component (i.e., class or aspect). If the concern is not tangled in any component, it is considered as *isolated*. A *scattered concern* spreads over multiple components. Our classification makes a distinction between *highly scattered* and *little scattered* concerns based on the number of affected components. A concern is *crosscutting* only if it is both tangled with other concerns and scattered over multiple system components. As we present later, even little scattered concerns might be considered as crosscutting in some scenarios. Crosscutting concerns generate warnings of inadequate separation of concerns and, consequently, opportunities for refactoring [10, 19].

Figure 2 presents definitions of rules (left) denoting transitions between two concern classifications (diagram on the right). This diagram makes it explicit the application order of the concern diffusion heuristics. This order was defined based on our empirical knowledge on analysing concern metrics [7, 12, 13, 22]. For instance, we usually check whether a concern is tangled or not (by means of NCC) before proceeding with further concern scattered analysis (by means of CDC). The first two rules, R01 and R02, use the metric Number of Concerns per Component (NCC) to classify the concern as isolated or tangled. If the NCC value is one for every component realising the analysed concern, it means that there is only the analysed concern in these components and, therefore, the concern is isolated. However, if NCC is higher than one in at least one component, it means that the concern is tangled with other concerns in that component, e.g., Factory Method and Observer patterns in MetaObjComposite (Figure 1).

Rules R03 and R04 (Figure 2) verify whether a concern, besides tangled, is scattered over multiple components. These heuristics use the metrics Concern Diffusion over Components (CDC) and Number of Components (NC) to calculate the percentage of system components affected by the concern of interest. Based on this percentage, the concern is classified as highly scattered or little scattered. As you might have already noticed, one of the most sensitive parts in a heuristic rule is the selection of threshold values. Our strategy for these rules is to use 50 % as default, but our tool (Section 5) allows customised threshold values. Note that, developers should be aware of highly scattered concerns because they can potentially cause design flaws, such as Shotgun Surgery [10] (Section 4.3).

R01 - *Isolated*:
if NCC = 1 for every component realising Concern
then Concern is Isolated

R02 - *Tangled*:
if NCC > 1 for at least one component then Concern is Tangled

R03 - *Little Scattered*:
if (CDC / NC) of Concern < 0.5 then Tangled Concern is Little Scattered

R04 - *Highly Scattered*:
if (CDC / NC) of Concern ≥ 0.5 then Tangled Concern is Highly Scattered

R05 - *Well Encapsulated*:
if (NOCA / NOA ≥ 0.5) and (NOCO / NOO ≥ 0.5) for every component
then Little Scattered Concern is Well-Encapsulated

R06 - *Crosscutting*:
if (NOCA / NOA < 0.5) and (NOCO / NOO < 0.5) for at least one component
then Little Scattered Concern is Crosscutting

R07 - *Well Encapsulated*:
if (NOCA / NOA ≥ 0.5) and (NOCO / NOO ≥ 0.5) for every component
then Highly Scattered Concern is Well-Encapsulated

R08 - *Crosscutting*:
if (NOCA / NOA < 0.5) and (NOCO / NOO < 0.5) for at least one component
then Highly Scattered Concern is Crosscutting

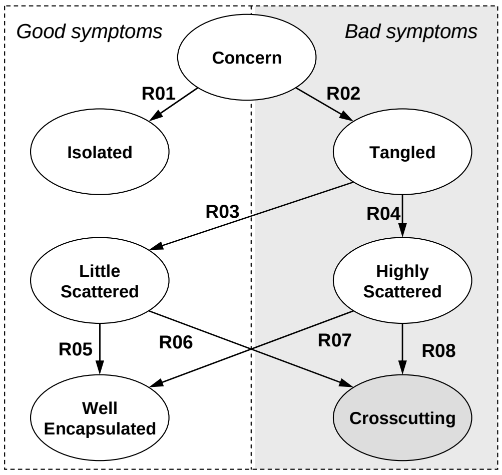


Figure 2. Concern classification and heuristics for concern diffusion

As tangling and scattering might mean different levels of crosscutting, both highly scattered and little scattered concerns can be classified as well-encapsulated or crosscutting concerns. Rules R05 and R06 decide whether a little scattered concern is either well-encapsulated or crosscutting. R07 and R08 perform similar analyses for a highly scattered concern. These four rules use the metrics Number of Concern Attributes (NOCA) and Number of Concern Operations (NOCO) and two size metrics (Table 1): Number of Attributes (NOA) and Number of Operations (NOO). They calculate for each component the percentage of attributes and operations which implements the concern being analysed.

The role of the heuristics R05 and R07 is to detect components that dedicate a large number of attributes and operations (more than 50%) to realise the dominant concern. If a concern is dominant in all components where it is, this concern is classified as well-encapsulated. The reasoning behind this classification is that a concern is not harmful to classes or aspects in which it is dominant, and, therefore, it does not need to be removed. Similarly, a concern is classified as crosscutting (rules R06 and R08) if the percentage of attributes and the percentage of operations related to the concern are low (less than 50%) in at least one component. Hence, a concern is classified as crosscutting if it is located in at least one component which has another concern as dominant.

4.2. Patterns of Crosscutting Concerns

Inspired by Ducasse, Girba and Kuhn [5], we defined in previous work [7] several patterns of crosscutting concerns (or *crosscutting patterns*). The identification of such crosscutting patterns is complementary to tangling and scattering analysis (Section 3.1). The reason is that recent studies have pointed out that certain shapes of tangled and scattered concerns are more harmful than others to design stability [7, 14]. Even though certain crosscutting patterns are recurring in software systems and likely to negatively affect software maintainability [7], there is no set of design rules defined for their identification. This section illustrates how the concern-sensitive heuristics can be used to identify five of these crosscutting patterns, namely Black Sheep, Octopus, God Concern, Data Concern, and Behavioural Concern. We have chosen them because they are the most primitive forms of crosscutting patterns [5, 7], and additional heuristics can be further defined to cover other crosscutting patterns through the specialisation of the primitive heuristics. The remainder of this section briefly describes each of these five selected crosscutting patterns and the associated heuristic rules to detect them.

Black Sheep. The Black Sheep crosscutting pattern is defined as a concern that crosscuts the system but is realised by very few elements in distinct components [5, 7]. For instance, Black Sheep occurs when only a few scattered operations and attributes are required to realise a concern in design artefacts. Also, there is not a single component whose main purpose is to implement this concern. Figure 3 (a) presents an abstract representation of the Black Sheep concern. The shadow grey areas in this figure indicate elements of the components (represented by boxes) realising the concern under consideration. Taking these four components into account, the Black Sheep instance is realised by few elements of two components.

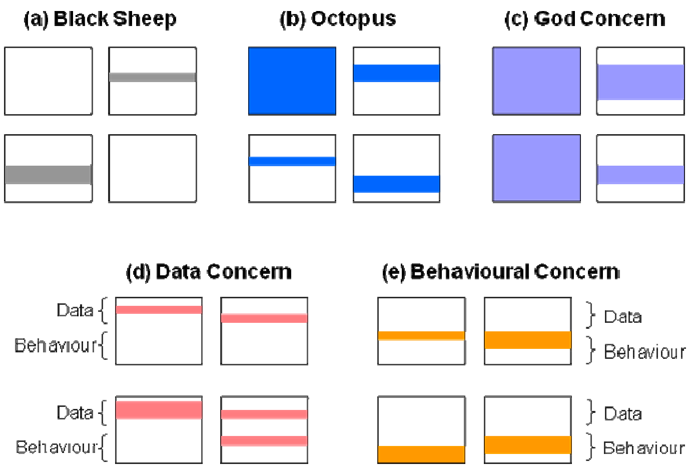


Figure 3. Abstract representation of crosscutting patterns

Figure 4 shows five heuristic rules, R09 and R13, which aim at identifying each of the selected crosscutting patterns. This figure also defines two conditions A (*Little Dedication*) and B (*High Dedication*) used by some of the rules. We explicitly separate the conditions from the rules not only to make the rules easier to understand but also to reuse the conditions in more than one heuristic rule. In these rules, a concern previously classified as crosscutting (Section 4.1) is thoroughly inspected in terms of its crosscutting patterns. A concern can be classified in more than one crosscutting pattern.

The heuristic rule R09 classifies a crosscutting concern as Black Sheep if all components which have this concern dedicate only a few percentage points of attributes and operations to that concern (less than 33 %). We choose the value of 33 % for the little dedication condition based on (i) threshold values used by other authors [5, 17], and (ii) a meaningful ratio that represents the definition '*touches very few elements*' of black sheep [5]. In fact, Lanza and Marinescu [17] suggest the use of meaningful threshold values, such as 0.33 (1/3), 0.5 (1/2), and 0.67 (2/3). Among these values, 1/3 seems the most appropriate one for the Black Sheep definition.

Condition A - *Little Dedication*: (NOCA / NOA < 0.33) and (NOCO / NOO < 0.33)

Condition B - *High Dedication*: (NOCA / NOA ≥ 0.67) and (NOCO / NOO ≥ 0.67))

R09 - *Black Sheep*:
if (*Little Dedication*) for every component with Concern then Crosscutting Concern is Black Sheep

R10 - *Octopus*:
if ((*Little Dedication*) or (*High Dedication*) for every component with Concern)
  and ((*Little Dedication*) for at least 2 components and (*High Dedication*) for at least 1 component with Concern)
then Crosscutting Concern is Octopus

R11 - *God Concern*:
if ((CDC / NC) > 0.33) and (High Dedication) for most components with Concern
then Crosscutting Concern is God Concern

R12 - Data Concern:
if ((NOCA/NOA) > (NOCO/NOO)) for every component with Concern
then Crosscutting Concern is Data Concern

R13 - Behavioural Concern:
if (NOCA = 0) and (NOCO > 0) for every component with Concern
then Crosscutting Concern is Behavioural Concern

Figure 4. Heuristics for crosscutting patterns

Octopus. Octopus is defined as a crosscutting concern which is partially well modularised by one or more components, but it is also spread across a number of other components [5, 7]. Figure 3 (b) presents an illustrative abstract representation of the Octopus crosscutting pattern. Note that, there is a component in this figure completely dedicated to the concern realisation (it represents the *octopus' body*). In addition, three other components (representing *tentacles*) dedicate only small parts to realise the same concern. The next rule presented in Figure 4, R10, verifies if a crosscutting concern is classified as Octopus. According to this heuristic, a concern is classified as octopus if every component realising parts of this concern has either little dedication (condition A) or high dedication (condition B) to the concern. Additionally, at least two components have to be little dedicated to the concern (tentacles of the octopus) and at least one component has to be highly dedicated (body of the octopus). We define a component as highly dedicated to a concern when the percentage of attributes and operations are higher than 67%. Again, thresholds values try to be a meaningful approximation of the octopus' definition [5, 7].

God Concern. Inspired by the God Class bad smell [20], the God Concern crosscutting pattern occurs when a concern encompasses too much scattered functionality of the system [7]. In other words, similarly to God Class, the elements realising God Concern address too much or do too much across the software system. The basic idea behind modular programming is that a widely-scoped concern should be analysed as smaller "modular" sub-parts (divide and conquer). However, God Concern instances do not adhere to this principle; thus, hindering concern analysis. Figure 3 (c) presents the abstract structure of this crosscutting pattern. This figure highlights that God Concern is a widely-scoped crosscutting concern and requires a lot of functionality in its realisation. In particular, there are at least two components whose main purpose is to realise this crosscutting concern.

The third heuristic rule in Figure 4, R11, aims at finding God Concern. A crosscutting concern is classified as God Concern if two conditions are satisfied: (i) the given concern is realised by more than 33% of the system's components and (ii) most of these components are highly dedicated to the concern realisation (Condition B). To verify the first condition, R11 calculates the percentage of components realising the given concern (i.e., CDC / NC). With respect to the second condition, not all components in the system are highly dedicated to the concern realisation, since this concern was previously classified as crosscutting (Section 4.1). In other words, this concern was previously classified as crosscutting in at least one component (see rules R06 and R08 in Section 4.1).

Therefore, we just need to verify if the concern is dominant (via the High Dedication condition) in more than 50% of the components where it is located.

Data Concern. The fourth crosscutting pattern, Data Concern, represents a crosscutting concern mainly composed of data [7]. Data may be represented, for instance, by class variables and fields in an object-oriented design. Data Concern has no (or very little) assigned behaviour. Behaviour can be associated with methods and constructors in an object-oriented design. In fact, existing behaviour in a Data Concern instance is due to the existence of data accessors (e.g., get and set methods). Figure 3 (d) shows the abstract representation of this crosscutting pattern. Note that, there is a logical division between data and behavioural elements of each box (i.e., component). Moreover, one component has a small piece of behaviour assigned to the analysed concern, which is accepted in the definition of this crosscutting pattern. However, most of the shadowed areas in the Data Concern instance represent data. The heuristic rule for Data Concern, R12 in Figure 4, is based on two concern metrics (NOCA and NOCO) two traditional size metrics (NOA and NOO). Then, the heuristic strategy is, for every component, to investigate if many attributes and few operations are realising the concern. Note that a concern can be classified as Data Concern even if some operations are used to realise it. That is, the Data Concern definition also allows some operations, such as getting and setting methods, realising a concern of this pattern. Moreover, we do not guarantee that the heuristic rule is 100% accurate and, therefore, some operations which are neither getting nor setting methods may be misled by this heuristic.

Behavioural Concern. The last crosscutting pattern, Behavioural Concern, identifies a concern only composed of behaviour (e.g., methods and constructors in an object-oriented design) [7]. Behavioural Concern has no associated data as illustrated by its abstract representation in Figure 3 (e). Figure 4 also shows a concern-sensitive heuristic rule, R13, aiming to identify Behavioural Concern. A concern is classified by R13 in this crosscutting pattern if it is only composed of operations. For programs written in Java, for instance, Behavioural Concern indicates a concern which is only realised by methods and constructors. On the other hand, advice may also be considered as an operation in AspectJ systems [25]. This decision, in fact, depends on the instantiation of specific language constructs to the metrics which composed a heuristic rule. To check whether a crosscutting concern is Behavioural Concern, R13 relies on the metrics NOCA and NOCO. Then, this rule verifies if the NOCA value is zero and NOCO is greater than zero for the analysed concern. If so, this crosscutting concern is classified as Behavioural Concern.

4.3. Concern-Aware Design Flaws

This section describes how the concern-sensitive heuristics could be also applied to detect well-known design flaws, such as bad smells [10, 20]. We use four bad smells to illustrate the concern heuristics: Shotgun Surgery, Feature Envy [10], Divergent Change [10], and God Class [20]. We selected these four bad smells because (i) previous work related them to crosscutting concerns [20] and (ii) they are representatives of three different groups of flaws discussed by Lanza and Marinescu [17]. Hence, addressing different categories of flaws allows us to correlate our results with their studies [17, 18]. These bad smells can be briefly defined as follows.

Shotgun Surgery occurs when a change in a characteristic (or concern) of the system implies many changes to a lot of different places [10].

Feature Envy is related to the misplacement of a piece of code, such as an operation or attribute, i.e., the feature seems more interested in a component other than the one it actually is in [10].

Divergent Change occurs when one component commonly changes in different ways for several different reasons [10]. Depending on the number of concerns a given component realises, it can suffer unrelated changes.

God Class refers to a component (class or aspect) which performs most of the work, delegating only minor details to a set of trivial components and using data from them [20].

Figure 5 shows concern-sensitive heuristic rules for detecting the four bad smells aforementioned. The first rule, R14, aims at detecting Feature Envy and uses a combination of concern metrics, coupling metrics, and size metrics. To be considered Feature Envy, a crosscutting concern has to satisfy two conditions: C (High Inter-Component Coupling) and D (Low Intra-Component Coupling). The concern under assessment has high inter-component coupling when its percentage of coupling (CSC/CBC) is higher than the percentage of internal component members ((NOCA+NOCO) / (NOA+NOO)) that realise this concern. In other words, the ratio of coupling to size (measured in terms of attributes and operations) of such crosscutting concern is higher than the same ratio for all other concerns in the same component. Similar computation is performed for identifying low intra-component coupling.

Condition C - *High Inter-Component Coupling*: (CSC / CBC) > ((NOCA+NOCO) / (NOA+NOO))

Condition D - *Low Intra-Component Coupling*: (ICSC / ((NOA+NOO)-(NOCA+NOCO))) < ((NOCA+NOCO) / (NOA+NOO))

R14 - *Feature Envy*:
if (*High Inter-Component Coupling*) and (*Low Intra-Component Coupling*) and (NCC > 1)
then Crosscutting Concern is Feature Envy

R15 - *Shotgun Surgery*:
if Concern is (*Tangled*) and (*Highly Scattered*) and (*Crosscutting*) then Crosscutting Concern is Shotgun Surgery

R16 - *Divergent Change*:
if (NCC > 3) and (CBC > 5) then Component has Divergent Change

R17 - *God Class*:
if (NCC >2) and (NOA > (NOAsystem/NC)) and (NOO > (NOOsystem/NC)) then Component is a God Class

Figure 5. Heuristic rules for bad smells

The following two heuristics for bad smells R15 and R16 (Figure 5) are intended to detect Shotgun Surgery and Divergent Change, respectively. Differently from the previous rules, R15 is composed of the outcomes from other heuristics. More precisely, a concern is classified as Shotgun Surgery if it was previously identified as *Tangled* (R02), *Highly Scattered* (R04), and *Crosscutting* (R08). Unlike Shotgun Surgery, which is related to concern scattering, Divergent Change is related to concern tangling. The heuristic rule for Divergent Change, R16, detects components with this bad smell if they realise more than 3 concerns each (NCC > 3). That is, there is a high degree of tangling in these components. In addition, a component suffering from Divergent Change also depends on many other components (CBC > 5). These default threshold values can be adjusted for specific projects based on the designer's experience.

Finally, a component is flagged as God Class by the last heuristic rule (R13) if, besides a high number of internal members (attributes and operations), it implements many concerns. R13 distinguishes high and low number of internal members by comparison with the average size of all other system components (NOAsystem/NC and NOOsystem/NC). This last rule uses a threshold value of 2 for the metric Number of Concern per Component (NCC). The reasoning is that a component implementing more than two concerns tends to aggregate too much unrelated responsibility.

5. ConcernMorph: Extensible Metrics-based Heuristic Analysis

Concern-sensitive heuristic analysis may be a tedious task without proper tool support. This section describes ConcernMorph[1], a tool to automatically apply the proposed concern-sensitive heuristic rules. ConcernMorph is implemented as an Eclipse plug-in and supports the heuristic rules presented in Section 4. The tool also implements a set of concern metrics (Section 3) used by the heuristic rules.

The ConcernMorph architecture. Concern analysis requires users to specify the projection of concerns into syntactic elements, such as classes, methods, and attributes. An existing tool, ConcernMapper [27], offers good support for this task and is used by ConcernMorph. In other words, our tool relies on the projection of concerns to syntactic elements provided by ConcernMapper. Figure 6 shows the architecture of ConcernMorph and its relationships to ConcernMapper and the Eclipse Platform. Both tools, ConcernMorph and ConcernMapper, are coupled to the Eclipse Platform. ConcernMorph defines two main modules (Figure 6): (i) *Metric Collector* which is responsible for computing concern metrics, and (ii) *Rule Analyser* which applies heuristic rules to classify crosscutting concerns (Section 4.1) and identify specific crosscutting patterns (Section 4.2). Even though the current implementation of ConcernMorph does not support bad smell detection yet, this only requires the extension of the metric collector with new metrics, and the specification of new design rules through the rule analyser.

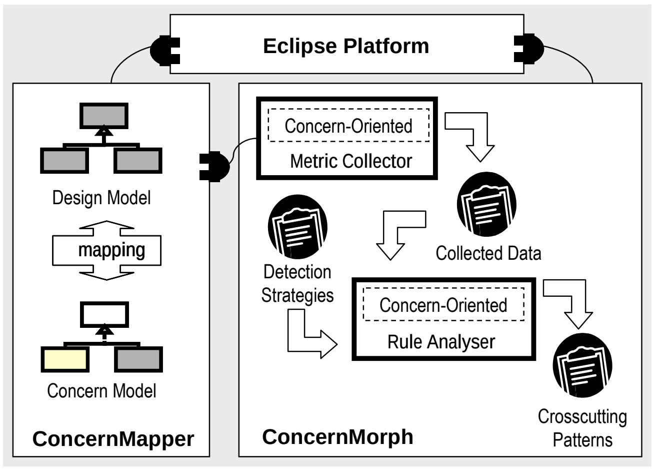
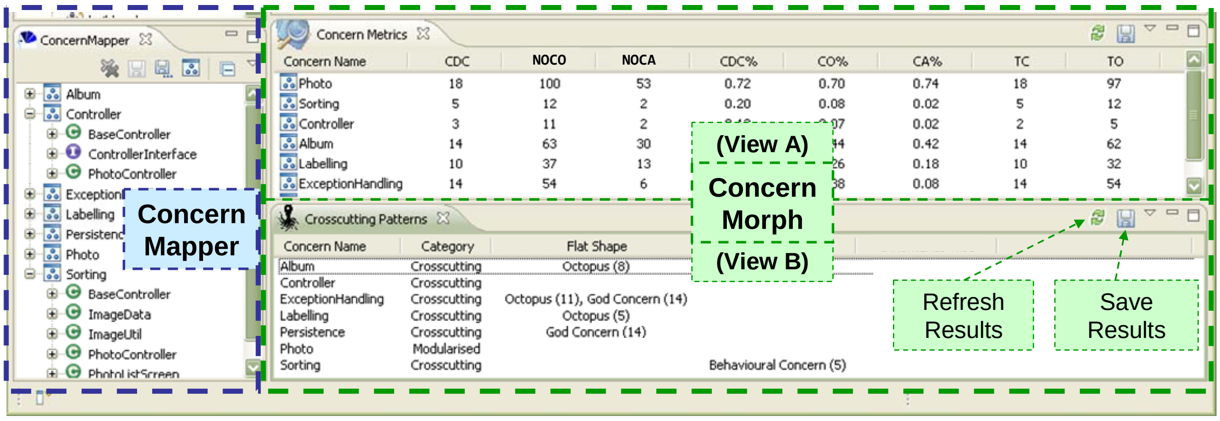


Figure 6. The ConcernMorph Architecture

The User Interface. Each module of ConcernMorph presented in Figure 6 adds a new view to the Eclipse Platform. Figure 7 shows the two views of ConcernMorph: Concern Metrics (View A) and Crosscutting Patterns (View B). The Concern Metrics view shows concern measurements while the Crosscutting Patterns view shows all instances of crosscutting patterns in the target system. In addition to two views, ConcernMorph extends ConcernMapper with an additional preference page. The ConcernMorph preference page supports the configuration of specific threshold values for the system under analysis. This page also allows the user to activate or deactivate a particular heuristic rule.

[1] *Available for download at http://www.lancs.ac.uk/postgrad/figueire/concern/plugin/*

We use one of our case studies, called MobileMedia, in the illustrative example of Figure 7. In the Concern Metrics view of this figure, we see that the Photo concern, for instance, is scattered over 18 components (CDC), 100 operations (CDO), and 53 Attributes (NOCA). The Crosscutting Patterns view shows not only all instances of crosscutting patterns but also the number of components (between brackets) taking part in each pattern instance. For example, the Album concern is classified as Octopus and 8 classes compose this Octopus instance (Figure 7).



- Album
- Controller
  - BaseController
  - ControllerInterface
  - PhotoController
- Exception
- Labelling
- Persistenc
- Photo
- Sorting
  - BaseController
  - ImageData
  - ImageUtil
  - PhotoController
  - PhotoListScreen

Concern Name	CDC	NOCO	NOCA	CDC%	CO%	CA%	TC	TO
Photo	18	100	53	0.72	0.70	0.74	18	97
Sorting	5	12	2	0.20	0.08	0.02	5	12
Controller	3	11	2	[illegible]	0.07	0.02	2	5
Album	14	63	30	[illegible]	44	0.42	14	62
Labelling	10	37	13	[illegible]	26	0.18	10	32
ExceptionHandling	14	54	6	[illegible]	38	0.08	14	54

Concern Name	Category	Flat Shape	
Album	Crosscutting	Octopus (8)	
Controller	Crosscutting		
ExceptionHandling	Crosscutting	Octopus (11), God Concern (14)	
Labelling	Crosscutting	Octopus (5)	
Persistence	Crosscutting	God Concern (14)	
Photo	Modularised		
Sorting	Crosscutting		Behavioural Concern (5)

Figure 7. Screenshot showing concern measurements (View A) and crosscutting patterns (View B).

6. Evaluation Settings

This section summarises the hypotheses and evaluation methodology of this paper . Section 6.1 formulates the hypothesis and research questions while Section 6.2 presents the seven applications used in our evaluation.

6.1. Hypothesis and Research Questions

We evaluated the accuracy and applicability of the heuristic assessment technique (Section 4) for identifying crosscutting concerns. In this particular case, we aimed to verify the following hypothesis.

> ***Hypothesis****: Heuristic assessment techniques which are based on concern properties enhance traditional quantitative analysis.*

To test this hypothesis, we followed three steps. First, we selected 23 concerns projected into implementation artefacts of six systems. Second, we used the concern-sensitive heuristics presented in Section 4.1 to heuristically identify crosscutting concerns. In addition to identify crosscutting concerns, the motivation of this evaluation was to verify if concern-sensitive heuristics minimise the shortcomings of traditional metrics-based strategies (discussed in the literature review, Section 2). Our hypothesis is that most of the problems of traditional strategies can be significantly ameliorated through the application of concern-sensitive analysis. Third, we evaluated whether the heuristic classification of crosscutting concerns indicates design flaws not reported by traditional software metrics. Additionally, we evaluated the accuracy of the heuristic identification of crosscutting concerns by comparing it to previous knowledge about the analysed concerns.

The following research questions are derived from the aforementioned hypothesis. Each subsection of Section 7 aims to answer one of these research questions.

Research Question 1*. If concern-sensitive assessment techniques properties enhance traditional quantitative analysis, which cases do they succeed?*

Research Question 2*. Do concern-sensitive heuristics provide accurate results compared to traditional detection strategies?*

Research Question 3*. Does a high number of crosscutting patterns impact positively, negatively or have no impact on design stability?*

Research Question 4*. Which bad smells can be detected by concern-sensitive heuristic rules?*

6.2. Selection of the Target Applications

In previous comparative studies [2, 9, 12-14], we applied AO metrics (e.g., metrics in Table 1, Section 4) to a number of software systems. In this paper, we selected seven systems of our previous studies in order to apply and assess the accuracy of our concern-sensitive heuristics. Five of these systems [2, 6, 8, 13, 15] are medium-sized academic prototypes carefully designed with modularity attributes as main drivers. The other two systems, namely Health Watcher [14] and the CVS plugin [9], are larger software projects. Since there are many releases available for some of these systems, we chose the 9th release of Health Watcher [14] and the 6th release of MobileMedia [8]. Table 2 summarises the seven studies, the nature of the concerns, and provides a complete list of the evaluated concerns. These studies were selected for several reasons. First, they encompass both AO and OO implementations and their assessments were previously based on metrics. They allowed us to evaluate whether concern-sensitive heuristics are helpful to enhance a design assessment exclusively based on metrics (Section 6.2). Furthermore, the modularity problems in all the selected systems have been correlated with external software attributes, such as reusability [9, 13, 15], design stability [2, 14], and manifestation of ripple effects [14].

Another reason for selecting theses systems is the heterogeneity of concerns found (2nd column, Table 2), which include widely-scoped architectural concerns, such as persistence and exception handling, and more localised ones, such as some design patterns. They also encompass different characteristics and different degrees of complexity. These systems are representatives of different application domains, ranging from simple design pattern instances (1st system) to reflective middleware systems (2nd), real-life Web-based applications (5th), multi-agent systems (6th), and software product lines (7th). Finally, such systems will also serve as effective benchmarks because they involve scenarios where the “aspectisation” of certain concerns is far from trivial [2, 9, 13, 14]. We took the analysed concerns from previous studies and the complete descriptions of such concerns and systems can be found in their respective publications (1st column of Table 2).

Table 2. Selected systems and respective concerns.

Systems	Nature of the Concerns	Concern Names
(1) A Design Pattern library [15]	Roles of design patterns	Director role (Builder), Handler role (Chain of Responsibility), Creator role (Factory Method), Subject and Observer roles (Observer), Colleague and Mediator roles (Mediator)
(2) OpenOrb Middleware [2]	Design patterns and their compositions	Factory Method, Observer, Facade, Singleton, Prototype, State, Interpreter, and Proxy
(3) AJATO [2, 6]		
(4) Eclipse CVS Plugin [9]	Recurring architectural crosscutting and non-crosscutting concerns	Business, Concurrency, Distribution, Exception Handling, and Persistence
(5) Health Watcher, R9 [14]		
(6) Portalware [13]	Domain-specific concerns	Adaptation, Autonomy, and Collaboration
(7) MobileMedia, R6 [8]	Product line features	Controller, Sorting, Exception Handling, Favourites, Persistence, and Labelling

7. Results and Discussion

This section introduces a systematic evaluation of the concern-sensitive heuristics in seven applications (Section 6.2) implemented in Java (OO) and AspectJ (AO). We applied the heuristic rules to 23 concern instances of both OO and AO versions of these applications. The concerns were selected because they exercise different heuristics. The heuristics evaluation is undertaken under four dimensions: (i) applicability for addressing measurement limitations (Section 7.1); (ii) accuracy to identify crosscutting concerns in both OO and AO systems (Section 7.2); (iii) ability to pinpoint sources of design instability (Section 7.3) and (iv) the usefulness to detect design flaws in comparison with conventional heuristics (Section 7.4). Section 7.5 discusses threats to the validity of our evaluation.

7.1. Solving Measurement Shortcomings

The goal of this section is to discuss whether and how concern-sensitive heuristics address the limitation of conventional modularity assessment. To achieve this goal, we used our tool (Section 5) to apply the proposed heuristic rules to the selected systems (Section 6.1). Table 3 provides an overview of the results for the first two suites of rules, namely concern diffusion (Section 4.1) and crosscutting patterns (Section 4.2). The 1st and 2nd columns in this table show the seven system and all analysed concerns, respectively. The columns labelled 01 to 13 present the answers 'yes' (y) or 'no' (n) of the heuristics for concern diffusion (rules 01-08) and crosscutting patterns (rules 09 to 13). Blank cells mean that the rule is not applicable. Finally, the last column indicates how the heuristics classify each concern. The obtained heuristic classification of concerns includes isolated, well-encapsulated ('well-enc.'), crosscutting, black sheep (BS), octopus (Oct), god concern (GC), data concern (DC), and behavioural concern (BC).

Based on these results, this section presents and discusses below six of problems (labelled A to F) that affect not only conventional metrics but also the use of concern metrics (Section 3). It also classifies these problems into four main categories. For simplicity reasons, we do not show all the results of the conventional and concern metrics for all systems, but we made the complete raw data available in a supplementary website [1].

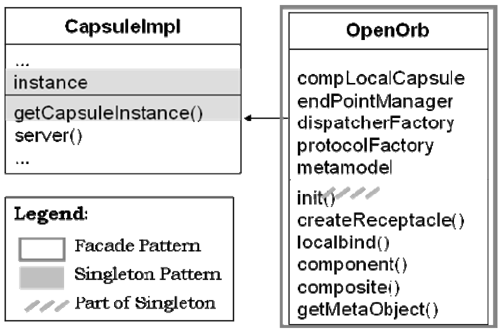
Table 3. Results of the heuristics for concern diffusion and crosscutting patterns in the OO systems.

Studies	**Concerns**	**Heuristic Rules**													**Concern Classifications**
		01	**02**	**03**	**04**	**05**	**06**	**07**	**08**	**09**	**10**	**11**	**12**	**13**	
(1) Patterns library [15]	Director role	N	y	y	n	y	n								well-enc.
	Handler role	N	y	n	y			y	n						well-enc.
	Creator role	N	y	n	y			n	y	n	n	n	n	y	BC
	Observer role	N	y	n	y			n	y	n	y	n	n	y	Oct + BC
	Subject role	n	y	n	y			n	y	n	y	n	n	n	Oct
	Colleague role	n	y	n	y			n	y	n	y	n	n	n	Oct
	Mediator role	n	y	n	y			y	n						well-enc.
(2) OpenOrb Middleware [2]	Fact. Method	n	y	y	n	y	n								well-enc.
	Observer	n	y	y	n	n	y			n	y	n	n	n	Oct
	Façade	y	n												isolated
	Singleton	n	y	y	n	n	y			y	n	n	n	n	BS
(3) Measurement tool [2, 6]	Prototype	n	y	y	n	n	y			y	n	n	n	y	BS + BC
	State	n	y	y	n	y	n								well-enc.
	Interpreter	n	y	y	n	n	y			n	y	n	n	n	Oct
	Proxy	n	y	y	n	n	y			n	n	n	n	n	crosscutting
(4) CVS plugin [9]	Exc. Handling	n	y	n	y			n	y	n	y	n	n	y	Oct + BC
(5) Health Watcher, R9 [14]	Business	n	y	n	y			y	n						well-enc.
	Concurrency	n	y	n	y			n	y	n	y	n	n	n	Oct
	Distribution	n	y	n	y			n	y	n	y	n	n	n	Oct
	Persistence	n	y	n	y			n	y	n	y	y	n	n	Oct + GC
(6) Portalware [13]	Adaptation	n	y	y	n	n	y			n	n	n	n	n	crosscutting
	Autonomy	n	y	y	n	n	y			n	n	n	n	n	crosscutting
	Collaboration	n	y	n	y			n	y	n	y	n	n	n	Oct
(7) MobileMedia, R6 [8]	Controller	n	y	n	y			n	y	n	y	n	n	n	Oct
	Sorting	n	y	y	n	n	y			y	n	n	n	n	BS
	Exc. Handling	n	y	n	y			n	y	n	y	n	n	n	Oct
	Favourites	n	y	y	n	n	y			y	n	n	n	n	BS
	Persistence	n	y	n	y			n	y	n	y	y	n	n	Oct + GC
	Labelling	n	y	n	y			n	y	n	y	n	n	n	Oct

False crosscutting warnings. We identified in this study at least two examples of problems in this category: *(A) false scattering and tangling warnings* and *(B) false coupling or cohesion warnings*. Problem A occurs when concern metrics erroneously warn the developer of a possible crosscutting concern. However, a subsequent careful analysis shows that the concern is well encapsulated. Similarly, Problem B leads developers to an unnecessary search for design flaws, but, in this situation, the false warning is caused by traditional metrics, such as coupling and cohesion ones. Figure 1 (Section 2) presents an example of this problem category in an instance of the Factory

Method pattern [11]. OO abstractions provide adequate means of modularising Factory Method (e.g., the main purpose of classes which implement this pattern is to realise it). However, the values of the concern metrics show some level of scattering and tangling in the classes realising this pattern (tables in Figure 1). In this case, the false warning was a result of the Observer-specific concerns crosscutting classes of the Factory Method pattern, i.e., it is not a problem of this latter pattern implementation at all. Our case studies indicate that shortcomings of this category are ameliorated with concern-sensitive heuristic support. For instance, Problem A discussed above (Factory Method) does not produce a false warning when the heuristic rules are applied (Table 2).

Hiding concern-sensitive flaws. Sometimes, design flaws are hidden in measurements just because *(C) metrics are not able to reveal an existing modularity problem*. We illustrate this limitation of metrics in the light of a partial class diagram presented in Figure 8. This figure highlights elements that implement the Singleton and Facade design patterns. It also shows some concern metrics for these patterns. Although Singleton has a lower average metric value than Facade, the former presents a crosscutting nature and the latter does not. Therefore, in this example, concern metrics do not warn the developer about the crosscutting phenomena relative to Singleton [2, 12]. The application of concern heuristics overcomes this measurement shortcoming by correctly classifying the Singleton pattern as black sheep – a specialised category of crosscutting concerns.



Pattern	CDC	NOCA	NOCO
Façade	1	5	6
Singleton	2	1	1

Component	NCC
CapsuleImpl	1
OpenOrb	2

Figure 8. Concern metrics for Facade and Singleton.

Lack of mapping to flaw-causing concerns. Two examples of problems were identified in this study: *(D) measurement does not show where (which design elements) the problem is* and *(E) measurement does not relate design flaws to concerns causing them*. One example of Problem D is the Collaboration concern of the Portalware system which is high scattered and tangled, e.g., CDC metric is 15 (components in Table 4). Nevertheless, in a more detailed analysis we found out that only six components have tangling among Collaboration and other system concerns. That is, the top six components listed in Table 4. Hence, focusing on Collaboration the developer needs to inspect (and perhaps improve) the design of only six components (and not 15). In order to solve this problem, the concern-sensitive heuristics classified Collaboration as octopus (Table 3). Furthermore, Rule 10 identified that the octopus' tentacles only touch six components which are indeed crosscut by Collaboration. The remaining 9 components well encapsulate Collaboration.

Controversial outcomes from concern metrics. A problem of this category occurs if *(F) the results of different metrics do not converge to the same outcome*, making the designer's interpretation difficult. We have identified some occurrences of this problem in our study, like the Adaptation concern (Portalware). Applying the concern metrics to Adaptation, the CDC value is 3 (indicating low scattering) while other concern metrics present high values (e.g., NOCA=10 and NOCO=22) [1]. Hence, the concern metrics have contradictory values in the sense that it is hard to conclude whether Adaptation is a crosscutting concern. The concern-sensitive heuristics addressed this category of shortcomings in a number of cases. For instance, although Adaptation has contradictory results for concern metrics, the heuristics have successfully identified it as an octopus (Table 3), meaning that it is also a crosscutting concern.

Table 4. Traditional and concern metrics for the Collaboration concern of Portalware.

	NOA	NOCA	NOO	NOCO
SearchingPlan	0	0	2	0
SearchResultReceivingPlan	0	0	2	0
AvailabilityPlan	0	0	2	0
SearchAskAnsweringPlan	0	0	2	0
ContentDistributionPlan	0	0	2	0
ResponseReceivingPlan	0	0	2	0
Collaboration	0	0	7	7
CollaboratorCore	2	2	11	11
CollaboratorRole	1	1	8	8
CollaborativeAgent	1	1	6	6
SharedObject	4	4	6	6
Answerer	1	1	4	4
Caller	1	1	3	3
ContentSupplier	1	1	2	2
Editor	0	0	4	4
Summation	11	11	63	51

7.2. Accuracy of the Concern-Sensitive Heuristics

This section evaluates the heuristics accuracy comparing their outcomes with the best means of modularisation (the OO or AO decomposition) based on specialists and literature claims [2, 9, 12-15]. Specialists are researchers that participated in development, maintenance, or assessment of the target systems. Their opinions were acquired by asking them to fill a questionnaire (available at [1]). In this step of our evaluation, in addition to the original designs, we also discuss the results for the aspectual ones. The aspectual implementations aim at modularising one or more crosscutting concerns.

Table 5 presents the results of this analysis for the concern diffusion heuristics (Section 4.1). We focus on the heuristics for concern diffusion because most of the specialists (e.g., developers who answered our questionnaire) are not familiar with the crosscutting patterns. Therefore, it would be hard to verify from this perspective the correctness of the heuristic rules for crosscutting pattern detection. The first two columns of this table are the same presented in Table 3. We show these columns just to facilitate the analysis of the results. The 3rd column presents the specialists' opinion or the best known solution (OO or AO). The results of the heuristic rules are given in the 4th and 5th columns, labelled Concern Classification. Detailed data for the heuristics application in the AO systems are available in our website [1].

Table 5. Results of the heuristics for concern diffusion in the OO and AO systems.

Studies	Concerns	Best Solution	Concern Classification		Results	
			OO	AO	OO	AO
(1) Patterns library [15]	Director role	OO	well-enc.	well-enc.	hit	hit
	Handler role	AO	well-enc.	well-enc.	FAIL	hit
	Creator role	OO	crosscutting	well-enc.	FAIL	hit
	Observer role	AO	crosscutting	crosscutting	hit	FAIL
	Subject role	AO	crosscutting	crosscutting	hit	FAIL
	Colleague role	AO	crosscutting	crosscutting	hit	FAIL
	Mediator role	AO	well-enc.	crosscutting	FAIL	FAIL
(2) OpenOrb Middleware [2]	Fact. Method	OO	well-enc.	isolated	hit	hit
	Observer	AO	crosscutting	isolated	hit	hit
	Façade	OO	isolated	isolated	hit	hit
	Singleton	AO	crosscutting	isolated	hit	hit
(3) Measurement tool [2, 6]	Prototype	AO	crosscutting	isolated	hit	hit
	State	AO	well-enc.	isolated	FAIL	hit
	Interpreter	AO	crosscutting	isolated	hit	hit
	Proxy	AO	crosscutting	isolated	hit	hit
(4) CVS plugin [9]	Exc. Handling	AO	crosscutting	crosscutting	hit	hit *
(5) Health Watcher, R9 [14]	Business	OO	well-enc.	well-enc.	hit	hit
	Concurrency	AO	crosscutting	isolated	hit	hit
	Distribution	AO	crosscutting	well-enc.	hit	hit
	Persistence	AO	crosscutting	well-enc.	hit	hit
(6) Portalware [13]	Adaptation	AO	crosscutting	well-enc.	hit	hit
	Autonomy	AO	crosscutting	crosscutting	hit	FAIL
	Collaboration	AO	crosscutting	isolated	hit	hit
(7) MobileMedia, R6 [8]	Controller	AO	crosscutting	crosscutting	hit	hit **
	Sorting	AO	crosscutting	isolated	hit	hit
	Exc. Handling	AO	crosscutting	crosscutting	hit	hit *
	Favourites	AO	crosscutting	isolated	hit	hit
	Persistence	AO	crosscutting	crosscutting	hit	hit **
	Labelling	AO	crosscutting	crosscutting	hit	hit **

** Exception Handling was only partially aspectised in the CVS plugin and MobileMedia.*
*** Controller, Persistence, and Labelling were not aspectised in MobileMedia.*

Of course, since the implementations of the concerns are different (object-oriented vs. aspect-oriented versions), the heuristic rules may give different results. For instance, the Creator role (Design Patterns Library) is classified as crosscutting in the object-oriented design and as well-encapsulated in the aspect-oriented one. Finally, the last two columns indicate if the heuristic classification matches with the best proposed solution ('hit') or not ('fail'). For this analysis, we quantify how many times the heuristics match previous knowledge (hits) and how many times they do not match (false positives or false negatives). A false positive is a case where the concern heuristics answered 'yes' while it should have been 'no', i.e., where they erroneously reported a warning. On the other hand, a false negative is the case where the rules answered 'no' while it should have been 'yes'; thus, where they failed to identify a design flaw.

Tables 6 provides overviews of the hits, false positives (FP) and false negatives (FN) of the rules for the 58 concern instances involved in this experiment (29 in OO and 29 in AO designs). The rows of this table are organised in three parts: the object-oriented instances (OO), the aspect-oriented instances (AO), and the general data of both paradigms (OO+AO). Each row describes the absolute numbers and its percentage in relation to the total number of concerns. According to data in Table 6, the heuristics failed in less than 20% of the cases (6 false positives and 3 false negatives). Focusing on the OO designs, we found out 1 case false positive and 3 cases of false negative. The false positive occurs with the Creator role of the Factory Method pattern (Table 5). In this pattern, the GUIComponentCreator class which plays the Creator role has also GUI-related elements, such as the showFrame() method [15]. Our conclusion is that, although the Factory Method pattern is not a crosscutting concern [12, 15], this particular instance presents a problem of separation of concerns. Therefore, the concern-sensitive heuristics were able to detect a problem which has not been reported by the specialists.

Table 6. Statistics for concern diffusion heuristics.

Studies	Hits (%)	FP (%)	FN (%)	Total (%)
OO	25 (86.2)	1 (3.5)	3 (10.3)	29 (100)
AO	24 (82.8)	5 (17.2)	0 (0.0)	29 (100)
OO + AO	48 (84.5)	6 (10.3)	3 (5.2)	58 (100)

Simple Program Slices. The heuristics also presented 3 instances of false negatives: (i) Chain of Responsibility (CoR) pattern, (ii) Mediator role of the Mediator pattern, and (iii) State pattern. The CoR pattern was not detected as a crosscutting concern because the pattern instance is too simple (due to its library nature [15]) in order to expose the pattern's crosscutting nature [2, 12]. In fact, classes playing the Handler role have only three members: the successor attribute, the handleClick() method, and one constructor. Since the first two realise the CoR pattern, the heuristics classify this concern as well-encapsulated (i.e., the main purpose of all components). A similar situation occurs with the Mediator design pattern (Table 2).

Selection and Granularity of Concerns. Moreover, the State design pattern is a false negative in the third case study due to the assessment strategy of using patterns, instead of roles, as concerns. Although State has two roles (Context and State), it was considered as a single concern in the third application. Additionally, the state transitions (part of State role) occur in classes playing the Context role. In other words, one role crosscuts only the other role, and the concern heuristics were unable to expose the pattern as a crosscutting concern. Hence, our conclusion for OO designs is that finer grained concerns, such as roles of design patterns, enable higher precision of the heuristics.

Table 3 also presents five instances of false positive in the heuristic assessment of aspectual systems. Similarly to State discussed above, recurring false positives occur in AO designs when a pattern defines more than one role. For instance, four (out of five) false positives match this criterion: Subject and Observer roles (the Observer design pattern) and Mediator and Colleague roles (the Mediator design pattern). Although each of these patterns is successfully modularised using abstract protocol aspects and concrete aspects, their roles share these components. Then, the Observer and Mediator roles were classified as crosscutting in the aspect protocol while their counterparts (Subject and Colleague) were classified as crosscutting in the concrete aspects. Therefore, differently from OO designs, our analysis suggests selecting the whole pattern (instead of its roles) for AO designs. In other words, the selection of the assessment granularity has to match the granularity of the aspects. So, since design patterns were aspectised, they should represent the granularity of the assessed concerns.

Aspect Precedence. The Autonomy concern is also misclassified as crosscutting in the AO design of Portalware. This situation occurs due to a precedence definition between the Adaptation and Autonomy aspects. The Adaptation aspect has a *declare precedence* clause with a reference to Autonomy. The concern heuristics point out Autonomy code inside the Adaptation aspect and, therefore, indicate that the former aspect crosscuts the latter. Although this problem really exists, there is no way of improve the separation of these two concerns in the Portalware system due to AspectJ constraints [14].

7.3. Correlating Crosscutting Patterns and Design Stability

This section analyses the correlation between the number of crosscutting patterns and changes in a component. In this analysis, we selected two systems, namely MobileMedia and Health Watcher, presented in Section 6.1. This part of the evaluation comprises three steps. First, we identified the most unstable components by analysing real changes throughout the MobileMedia and Health Watcher software evolutions. We extracted these changes from 7 successive releases of MobileMedia and 10 releases of Health Watcher. Second, we heuristically detected crosscutting patterns by analysing concerns in the MobileMedia and Health Watcher designs (Section 6.2). Finally, we contrasted the number of crosscutting pattern instances found in each component with the list of unstable components (first step).

Table 7. Unstable MobileMedia components and crosscutting patterns (Release 6).

Components	**Crosscutting Patterns**					**Total of Patterns**	**Total of Changes**
	BS	**Oct**	**GC**	**DC**	**BC**		
BaseController	0	4	1	0	0	5	5
MediaAccessor	0	3	1	0	0	4	5
MediaController	2	4	1	0	0	7	5
MediaUtil	2	4	1	0	0	7	5
PhotoViewScreen	0	3	0	0	0	3	4
<Complete list is at [1]>							
SmsReceiverThread	0	2	0	0	0	2	1
SmsSenderThread	0	2	0	0	0	2	1
UnavailablePhotoAlbumException	0	2	0	0	0	2	1
BaseThread	0	0	0	0	0	0	0
SplashScreen	0	0	0	0	0	0	0
Average	**0.3**	**2.9**	**0.7**	**0.0**	**0.0**	**3.8**	**1.8**
Correlation	**41.2 %**	**40.5 %**	**27.6 %**	**0.0%**	**0.0%**	**48.5%**	

Table 7 illustrates our results of MobileMedia by showing 5 of the most unstable components (top components) and 5 of the most stable components (bottom components) for the 6th object-oriented release. This analysis focuses on the object-oriented design because crosscutting patterns revealed to be more common in this type of software decomposition technique [7]. These tables also show the number of changes (last column) and the number of crosscutting patterns (intermediary columns) per component of MobileMedia. Considering that a system evolves towards a more stable structure, we focused on Release 2 to represent a later (more stable) design. Table 8 presents similar information about crosscutting patterns and design stability for the object-oriented design of Health

Watcher. We focus our discussion here on these representative MobileMedia and Health Watcher releases, but additional data are presented at [1].

Table 8. Unstable Health Watcher components and crosscutting patterns (R09).

Components	Crosscutting Patterns					Total of Patterns	Total of Changes
	BS	Oct	GC	DC	BC		
HealthWatcherFacade	0	3	1	0	0	4	6
HWServlet	0	1	1	0	0	2	6
UpdateHealthUnitSearch	0	3	1	0	0	4	6
UpdateComplaintData	0	2	1	0	0	3	5
GetDataForSearchByDiseaseType	0	2	1	0	0	3	4
<Complete list is at [1]>							
Subject	0	0	0	0	0	0	0
SymptomRecord	0	3	1	0	0	4	0
ThreadLogging	0	0	0	0	0	0	0
TransactionException	0	3	0	0	0	3	0
UpdateEntryException	0	1	0	0	0	1	0
Average	**0**	**1.5**	**0.5**	**0**	**0**	**2.0**	**1.1**
Correlation	**0**	**31.1 %**	**47.9 %**	**0**	**0**	**46.9%**	

Correlating Crosscutting Patterns and Changes. From data of Tables 7 and 8, it is easy to recognise that classes with higher occurrences of changes are also the locus of more instances of crosscutting patterns. For instance, classes of MobileMedia (Table 7) changing in 5 out of 7 releases are affected by at least 4 instances of crosscutting patterns. On the other hand, usually no more than 2 instances of crosscutting patterns could be found in the most stable classes of both systems, with just two exceptions.

In our analysis, we also performed the Spearman's correlation [31] between crosscutting patterns and design changes as presented in the last rows of Tables 7 and 8. Additionally, we used the following criteria to interpret the strength of correlation [31]: less than 10% means trivial, 10% to 30% means minor, 30% to 50% means moderate, 50% to 70% means large, 70% to 90% means very large, and more than 90% means almost perfect. This indicator shows a moderate correlation between crosscutting patterns and design in both analysed systems. Based on these preliminary results, we observed that the proposed heuristic rules for crosscutting patterns detection can be used as intuitive indicators of design instability.

Analysis of Specific Crosscutting Patterns. We also investigated the correlation of specific crosscutting patterns and design stability. In order to investigate how each crosscutting pattern relates to design stability, we quantified per component (i) the number of changes and (ii) the number of individual crosscutting patterns. This analysis is based on the results of the Spearman's correlation presented in the last row of Tables 7 and 8. As it can be observed in these tables, we could not find instances of Data Concern and Behavioural Concern in the two target systems (MobileMedia and Health Watcher). Therefore, our analysis focuses on the other three crosscutting patterns, Black Sheep, Octopus, and God Concern.

Interestingly, data in Tables 7 and 8 indicate that, when instances of the analysed crosscutting patterns can be found in a system, they usually present moderate correlation with module changes (i.e., between 30% and 50% correlation). Data of Octopus in Tables 7 and 8, for instance, presented a correlation of 40% and 31% in MobileMedia and Health Watcher, respectively. Minor correlation was only found for God Concern in the MobileMedia analysis. However, the correlation of God Concern and components changes was moderate in Health Watcher. Although preliminary, our results indicate that, even when used in isolation, a heuristic rule for one specific crosscutting pattern may be a valuable indicator of design instability.

7.4. Specific Design Flaws Detection

This section evaluates whether our concern heuristics are useful to detect specific design flaws. In particular, we applied rules R14 to R17 (Figure 5) to identify instances of four bad smells - Shotgun Surgery [10], Feature Envy [10], Divergent Change [10], and God Class [20] - in the seven target systems (Section 6.1). We also independently applied conventional heuristic rules proposed by Marinescu [17, 18] for detecting the same bad smells. The application of each rule pointed out design fragments suspect of having one of the three aforementioned bad smells. We then compared the results from the two suites of rules (concern-sensitive and conventional) with a reference list which indicates the actual bad smells in these systems.

The Bad Smells Reference List. We relied on two specialists of each system to build a reference list for each analysed bad smell; resulting in four reference lists for each system. Specialists were instructed to individually use their own strategy and knowledge about the systems to detect "smelly" classes. Not surprisingly, the two sets containing potential bad smell instances of a system were not exactly the same, although in some cases they have many classes in common. In order to reach an agreement, we undertook manual investigation in all suspect design fragments. The manual investigation allowed us to verify whether the suspect design fragments were indeed affected by the design flaw.

After this inspection, the bad smells indicated by heuristic rules were classified into two categories: (i) 'Hits': those heuristically suspect fragments that have confirmed to be affected by the bad smell, and (ii) 'False Positives': those heuristically suspect fragments that revealed not to be affected by the bad smell. Table 9 summarises the results of applying both concern-sensitive heuristics and conventional ones. This table shows the total number of hits and false positives (FP) for each bad smell and the percentage (under brackets) regarding the total number of suspect fragments. Note that the results in Table 9 are given in different view points. In concern-sensitive rules, the values for Shotgun Surgery and Feature Envy (R11 and R12) represent the number of concerns (since R11 and R12 classify concerns). On the other hand, God Class, Divergent Change, and conventional rules present the results in terms of the number of suspected classes. Moreover, the conventional heuristic rules [17, 18] do not support the detection of Divergent Change.

Table 9. Statistics for the heuristics related to bad smells.

Bad Smell	**Concern-Sensitive Rules**		**Conventional Rules**	
	Hits (%)	**FP (%)**	**Hits (%)**	**FP (%)**
Shotgun Surgery	10 (76.9%)	3 (23.1%)	9 (47.4%)	10 (52.6%)
Feature Envy	6 (85.7%)	1 (14.3%)	2 (25.0%)	6 (75.0%)
Divergent Change	4 (57.1%)	3 (42.9%)	*Not Supported*	
God Class	9 (81.8%)	2 (18.2%)	2 (25.0%)	6 (75.0%)

Data of Table 9 show that concern-sensitive heuristics presented superior accuracy than conventional rules for detecting all studied bad smells. The former presented no more than 25% of false positives, whereas the latter fails in more than 50% of the cases. We could also verify that this advantage in favour of our rules was mainly due to the fact that they are sensitive to the design concerns. For instance, many false positives of the conventional rule for Shotgun Surgery occurred because its metrics do not distinguish coupling between classes of the same concern and classes with different concerns.

Based on these values, we observed that the types of code smells that benefit most from the use of the concern-sensitive heuristic rules were Feature Envy and God Class. An example is the BaseController class in the MobileMedia system which was identified by the concern-sensitive heuristics for both God Class and Feature Envy bad smells. This class was not indicated for any bad smell by conventional heuristics. The main reason for the concern-sensitive heuristics to detect these bad smell instances is because this class is an outlier both in terms of its size and in terms of the number of concerns it realises. For instance, BaseController is a large class and realises 5 different concerns. Therefore, it was considered a God Class instance in the reference list. Additionally, some of its methods, such as showImageList and handleCommand, were considered in the wrong place and, therefore they should be moved to a different class. This situation indicates a Feature Envy instance.

7.5. Study Constraints

The goal of this paper was to assess how concern-sensitive heuristics can reduce false positives and false negatives of metrics and conventional heuristics. However, we have not assessed the reliability and feasibility of concern identification. For instance, identifying the Exception Handling concern in the source code is almost 100% automated because it is clearly marked in the systems by language constructs, such as try-catch blocks. On the other hand, concerns like Persistence, Distribution and Concurrency are more application dependents leading to a more complex identification process. Fortunately, recently proposed approaches and tools facilitate the concern mining [8, 21]. We also relied on specialists and literature claims. However, these can be themselves considered a threat to the validity of our study.

We evaluated the applicability of the approach by defining thirteen concern-sensitive heuristic rules that are classified in three categories. Based on their application to seven systems, we have shown that concern heuristics are accurate means of flaw detection and are usable in practice. However, we do not claim that our data are statistically relevant due to the reduced number of case studies. We have also not used rigorous statistical methods in all dimensions of the empirical evaluation. Nonetheless, we are considering the sample representative of the population due to the heterogeneity of systems and concerns involved in this study (Section 6.1).

The definition of proper threshold values intrinsically characterises any assessment approach [17, 18]. Our strategy based on our empirical evidence was to use: (i) 50% as default for the concern diffusion heuristics (Section 4.1), (ii) a meaningful ratio [17] that represents the definition of black sheep and octopus [5] (Section 4.2), and similar thresholds used by Marinescu [17, 18] in the heuristics for bad smells (Section 4.3). Of course those values might need to be varied depending on application particularities and assessment goals.

8. Concluding Remarks

This paper (i) presented a suite of concern-sensitive heuristic rules, and (ii) investigated the hypothesis that these heuristics offer enhancements over typical metrics-based assessment approaches. The heuristics suite can be extended and, by no means, we claim that the suite of rules proposed in this paper is complete. For instance, Section 4.2 presents heuristics for only five from a catalogue of 13 crosscutting patterns [7]. Furthermore, the heuristic suite for concern-aware design flaws (Section 4.3) focuses on bad smells and could be extended to deal with additional sorts of modularity flaws.

Our investigation indicated promising results in favour of concern-sensitive heuristics. First, the concern-sensitive heuristics performed well for determining concerns that should be aspectised. Our evaluation pointed out that the heuristics have around 80% of accuracy (Sections 5.3). Moreover, heuristics provided enhancements both in purely OO and in aspectual design assessment. Second, our investigation also pointed out that the presence of crosscutting patterns can be correlated with design instability (Section 6.4). It indicated that the classes that suffered a high number of changes during the systems' evolution were also locus of a high number of crosscutting patterns detected by our heuristic rules. Finally, our evaluation showed that the concern-sensitive rules were more accurate than conventional rules to detect four well-known bad smells (Section 6.5).

We also identified, however, some problems in the accuracy of the heuristics when (Section 6.3): (i) they were applied to small design slices (patterns library [15]), and (ii) there was a presence of concern overlapping. Similar problems remained when the heuristics were applied to the corresponding aspectual designs. For instance, some concerns were misclassified in scenarios involving an explicit precedence declaration between two or more aspects. However, even in presence of these intricate concern interactions (all systems, but the patterns library) the heuristics presented acceptable rates (Table 6).

References

[1] Figueiredo, E., Sant'Anna, C., Garcia, A. and Lucena, C., 2010. Applying and Evaluating Concern-Sensitive Design Heuristics. URL: http://www.dcc.ufmg.br/~figueiredo/concern/heuristics

[2] Cacho, N. et al., 2006. Composing Design Patterns: A Scalability Study of Aspect-Oriented Programming. In proceedings of the 5th International Conference on Aspect Oriented Software Development (AOSD), pp. 109-121. Bonn, Germany.

[3] Ceccato, M., and Tonella, P., 2004. Measuring the Effects of Software Aspectization. In proceedings of the 1st workshop on Aspect Reverse Engineering, paper 1, The Netherlands.

[4] Chidamber, S., and Kemerer, C., 2004. A Metrics Suite for Object Oriented Design. IEEE Transactions on Software Engineering, pp. 476-493.

[5] Ducasse, S., Girba, T., and Kuhn, A., 2006. Distribution Map. In proceedings of the International Conference on Software Maintenance (ICSM), pp. 203-212, Philadelphia, USA.

[6] Figueiredo, E., Garcia, A. and Lucena, C., 2006. AJATO: An AspectJ Assessment Tool. . In proceedings of the European Conference on Object-Oriented Programming (ECOOP), demo D9, Nantes, France.

[7] Figueiredo, E. et al., 2009. Crosscutting Patterns and Design Stability: An Exploratory Analysis. . In proceedings of the International Conference on Program Comprehension (ICPC), pp. 138-147, Vancouver, Canada.

[8] E. Figueiredo, et al., 2008. Evolving Software Product Lines with Aspects: An Empirical Study on Design Stability. In proceedings of the 30th International Conference on Software Engineering (ICSE), pp. 261-270. Leipzig, Germany.

[9] Filho, F. et al., 2006. Exceptions and Aspects: The Devil is in the Details. In proceedings of International Symposium on Foundations of Software Engineering (FSE), pp. 152-162. Portland, USA.

[10] Fowler, M., 1999. Refactoring: Improving the Design of Existing Code. Addison-Wesley, Reading, MA, USA.

[11] Gamma, E. et al., 1995. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, USA.

[12] Garcia, A. et al., 2006. Modularizing Design Patterns with Aspects: A Quantitative Study. Transactions on Aspect-Oriented Software Development, 1, pp. 36-74.

[13] Garcia, A. et al., 2004. Separation of Concerns in Multi-Agent Systems: An Empirical Study. Software Engineering for Multi-Agent Systems II, LNCS 2940, pp. 49-72.

[14] Greenwood, P. et al., 2007. On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study. In proceedings of European Conference on Object-Oriented Programming (ECOOP), pp. 176-200, Berlin, Germany.

[15] Hannemann, J., and Kiczales, G., 2002. Design Pattern Implementation in Java and AspectJ. In proceedings of the International Conference on Object-Oriented Programming, Systems, Language and Appications (OOPSLA), pp. 161-173, Seattle, USA.

[16] Kiczales, G. et al., 1997. Aspect-Oriented Programming. In proceedings of the European Conference on Object-Oriented Programming (ECOOP), LNCS 1241, Springer, pp. 220-242.

[17] Lanza, M., and Marinescu, R., 2006. Object-Oriented Metrics in Practice. Springer, USA.

[18] Marinescu, R., 2004. Detection Strategies: Metrics-Based Rules for Detecting Design Flaws. In proceedings of the International Conference on Software Maintenance (ICSM), pp. 350-359, Chicago, USA.

[19] Monteiro, M., and Fernandes, J., 2005. Towards a Catalog of Aspect-Oriented Refactorings. In proceedings of the International Conference on Aspect-Oriented Software Development (AOSD), pp. 111-122, Chicago, USA.

[20] Riel, A., 1996. Object-Oriented Design Heuristics. Addison-Wesley, Reading, MA, USA.

[21] Robillard, M., and Murphy, G., 2007. Representing Concerns in Source Code. ACM Transactions on Software Engineering and Methodology (TOSEM), 16, 1.

[22] Sant'Anna, C. et al., 2003. On the Reuse and Maintenance of Aspect-Oriented Software: An Assessment Framework. In proceedings of the Brazilian Symposium on Software Engineering (SBES), pp. 19-34, Manaus, Brazil.

[23] Zhao, J., 2002. Towards a Metrics Suite for Aspect-Oriented Software. Technical-Report SE-136-25, Information Processing Society of Japan (IPSJ).

[24] Sahraoui, H. Godin, R. and Miceli, T., 2000. Can Metrics Help to Bridge the Gap Between the Improvement of OO Design Quality and Its Automation? In proceedings of the International Conference on Software Maintenance (ICSM), IEEE Computer Society, pp. 154-162, San Jose, USA.

[25] Kiczales, G; Hilsdale, E; Hugunin, J; Kersten, M; Palm, J; Griswold, W., 2001. An overview of AspectJ. In proceedings of the European Conference on Object-Oriented Programming (ECOOP). Springer, pp. 327-354, Berlin, Germany.

[26] Mezini, M. and Ostermann, K., 2003. Conquering Aspects with Caesar. In proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD), pp. 90-99, Boston, Massachusetts.

[27] Robillard, M. and Weigand-Warr, F., 2005. ConcernMapper: Simple View-Based Separation of Scattered Concerns. In Proceedings of the Eclipse Technology Exchange at OOPSLA.

[29] Soares, S.; Laureano, E.; Borba., P., 2002. Implementing Distribution and Persistence Aspects with AspectJ. SIGPLAN Note, vol. 37, no. 11, pp. 174-190.

[30] Monteiro, M.; Fernandes, J., 2005. Towards a Catalog of Aspect-Oriented Refactorings. In proceeding of the International Conference on Aspect-Oriented Software Development (AOSD), pp. 111-122, Chicago, USA.

[31] Myers, J. and Well, A., 2003. Research Design and Statistical Analysis (2nd Edition), Lawrence Erlbaum.

[32] Marinescu, R. 2002. Measurement and Quality in Object-Oriented Design. PhD Thesis, Faculty of Automatics and Computer Science, Politehnica University of Timisoara.

[33] Briand, L.C.; Daly, J.W.; and Wüst, J.K. , 1999. A Unified Framework for Coupling Measurement in Object-Oriented Systems, IEEE Transactions on Software Engineering (TSE), vol. 25, pp. 91-121.

About the Authors

Eduardo Figueiredo is currently an assistant professor at Federal University of Minas Gerais in Brazil. He holds a PhD degree in Computer Science certified in 2009 by Lancaster University (U.K.). His research interest includes software metrics and heuristic analysis, empirical software engineering, software product lines, and aspect-oriented software development. Eduardo has consistently been publishing research papers in premier software engineering conferences, such as ICSE, ECOOP, and FSE. He has also co-organised two editions of the International Workshop on Assessment of Contemporary Modularization Techniques (ACoM) in 2007 and 2008 and the first edition of the Workshop on Empirical Evaluation of Software Composition Techniques (ESCOT) in 2010. http://www.facom.ufu.br/~figueiredo/

Claudio Sant'Anna is an assistant professor at the Federal University of Bahia (UFBA), Brazil. He received his PhD in computer science from the Pontifical Catholic University of Rio de Janeiro (PUC-Rio), Brazil on April 2008, in cooperation with Lancaster University, UK. His research interests include software metrics, empirical evaluation of contemporary modularization techniques, aspect-oriented software development, software design, and software architecture.

Alessandro Garcia has been an assistant professor in the Informatics Department at PUC-Rio, Brazil, since January 2009. He was a Lecturer in the Computing Department at Lancaster University, United Kingdom, from 2005 to 2009. He was previously a software architect at the IBM Almaden Research Center and a visiting researcher at the University of Waterloo, Canada. His main research interests are in the areas of exception handling, advanced modular programming, software architecture, software product lines, and empirical software engineering. He has been serving as a committee member of premier international conferences on software engineering, such as ICSE, AOSD, SPLC, OOPSLA/SPLASH, and AAMAS. He received many awards and distinctions, including the Best Dissertation Award (Computer Brazilian Society, 2000), Best Researcher Award (Lancaster University, 2006), Distinguished Young Scholar (PUC-Rio, 2009), Research Productivity Grant (CNPq, 2009), and Young Scientist Fellowship (FAPERJ). He has been involved in several projects, funded by EC and Brazilian research agencies, and his research contributions have been exploited by a number of industry partners, such as Siemens, SAP, IBM, and Motorola. http://www-di.inf.puc-rio.br/~afgarcia/

Carlos Lucena received a BSc degree from the Pontifical Catholic University of Rio de Janeiro (PUC-Rio), Brazil, in 1965, a MMath degree in computer science from the University of Waterloo, Canada, in 1969, and a PhD degree in computer science from the University of California at Los Angeles in 1974. He has been a full professor in the Departamento de Informatica at PUC-Rio since 1982. His current research interests include software design and formal methods in software engineering. He is member of the editorial board of the International Journal on Formal Aspects of Computing.