2009 XXIII Brazilian Symposium on Software Engineering

# Applying and Evaluating Concern-Sensitive Design Heuristics

Eduardo Figueiredo
Computing Department
Lancaster University
Lancaster, UK
e.figueiredo@lancaster.ac.uk

Claudio Sant'Anna
Computer Science Department
Federal University of Bahia (UFBA)
Salvador, Brazil
santanna@dcc.ufba.br

Alessandro Garcia, Carlos Lucena
Informatics Department, Opus Research Group - LES
PUC-Rio
Rio de Janeiro, Brazil
{afgarcia,lucena}@inf.puc-rio.br

*Abstract*— **Empirical studies have stressed that aspect-oriented decompositions can cause non-obvious flaws in the modularity of certain design concerns. Without proper design evaluation mechanisms, the identification of these flaws can become counter-productive and impractical. Nowadays, modularity assessment is mostly supported by metric-based heuristics rooted at conventional attributes, such as module cohesion and coupling. However, such conventional module-driven assessment cannot be tailored to the design concerns. This paper proposes and systematically evaluates a representative suite of concern-sensitive heuristic rules. The accuracy of the heuristics is assessed through their application to six systems. The analysis was based on the heuristics support for: (i) addressing the shortcomings of conventional metrics-based assessments, (ii) reducing the manifestation of false positives and false negatives, and (iii) finding the presence of design flaws relative to both classes and aspects.**

*Resumo*— **Estudos experimentais recentes mostraram que decomposições orientadas a aspectos podem causar anomalias na modularidade do design de certos interesses e que tais anomalias muitas vezes não são óbvias. Sem mecanismos de avaliação de design apropriados, a identificação dessas anomalias pode se tornar contraproducente e impraticável. Atualmente, a avaliação da modularidade de design orientado a aspectos é na maioria das vezes apoiada por heurísticas baseadas em métricas que quantificam atributos convencionais, como coesão e acoplamento de módulos. No entanto, essa avaliação dirigida por atributos convencionais não leva em conta os interesses que guiam o design. Esse artigo propõe e avalia sistematicamente um conjunto de regras heurísticas sensíveis a interesses. A acurácia das heurísticas foi avaliada por meio de sua aplicação em seis diferentes sistemas. A análise se baseou na capacidade das heurísticas de: (i) tratar das limitações de abordagens de avaliação baseadas em métricas convencionais, (ii) detectar a presença de anomalias de design relacionadas a classes e aspectos, e (iii) reduzir a manifestação de falsos positivos e falsos negativos.**

***Keywords-software design; modularity assessment; metrics; aspect-oriented software development***

## 1 INTRODUCTION

Aspect-oriented software development (AOSD) [16] is a recently proposed paradigm with the goal of enhancing design modularisation through new composition mechanisms. Aspects are new units of modularity for encapsulating crosscutting concerns, i.e., system features or properties that naturally affect many system modules [16]. However, the achievement of modular aspectual designs is far from being trivial as the separation of certain concerns with aspects can be harmful [9, 12]. Inaccurate concern modularisations can lead to multiple design flaws [9, 14]. They can promote violations of important concern-specific modularity principles, such as higher concern coupling and wider concern interfaces [2, 9, 12-14]. Even the "aspectisation" of conventional crosscutting concerns, such as exception handling [9], concurrency control [14], and the Observer design pattern [11, 15], might impose negative effects on the system modularity [2, 9, 14].

As a result, aspectual decompositions require proper mechanisms for detecting key categories of concern-driven design flaws, such as modularity anomalies [12, 17, 18]. Metrics and heuristics are traditionally the fundamental mechanisms for assessing design modularity [4, 17, 18, 20]. To date, aspect-oriented design assessment has been mostly rooted at extensions of module-level metrics [4, 22, 23] that have been historically explored in software engineering. 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 AOSD, there is a growing body of relevant work in the software engineering literature focusing either on concern representation and identification

978-0-7695-3844-0/09 $25.00 © 2009 IEEE
DOI 10.1109/SBES.2009.14

techniques [5, 7, 21] or on concern analysis tools [8, 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 enhances the process of evaluating aspectual designs.

In this context, the contributions of this paper are threefold. First, after revisiting existing assessment mechanisms, it discusses the limitations of conventional metrics-based heuristics (Section 2). Second, it presents a suite of heuristics 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 detecting overlapping categories of modularity problems, namely concern diffusion, crosscutting patterns [7], and classical design flaws [10, 20]. Third, this paper provides a systematic evaluation on the accuracy of the concern-sensitive heuristics in the context of six applications: three of them are medium-sized academic prototypes and the other three are larger software projects. We have analysed both OO and aspectual designs of such systems, which encompass heterogeneous forms of crosscutting and non-crosscutting concerns (Section 5). The overall result of our evaluation indicates that concern-sensitive heuristics (i) enhance most of the shortcomings of conventional assessment mechanisms, and (ii) present superior identification rates of modularity flaws compared to conventional heuristics.

## 2 Aspectual Design Assessment

This section discusses some metrics and conventional heuristics for modularity evaluation of aspect-oriented (AO) systems (Section 2.1). It also points out limitations of these conventional assessment mechanisms (Section 2.2).

### *2.1 Conventional Modularity Assessment*

A number of AO metrics [3, 22, 23] have been recently defined to quantify design modularity. However, most of these metrics are based on extensions of metrics for OO design assessment, such as Chidamber and Kemerer's Coupling between Objects (CBO) and Lack of Cohesion in Methods (LCOM) [4]. There are also some AO metrics based on dependence graphs [23] that capture different facets of coupling and cohesion. All these metrics are often rooted at attributes such as syntax-based module cohesion, coupling between modules, and module interface complexity.

Despite the extensive use of metrics, if used in isolation metrics are often too fine grained to quantify comprehensively an investigated modularity flaw [18]. In order to overcome this limitation of metrics, some researchers [17, 18] 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. To the best of our knowledge, all current design heuristics 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.

### *2.2 Limitation of Conventional Heuristics*

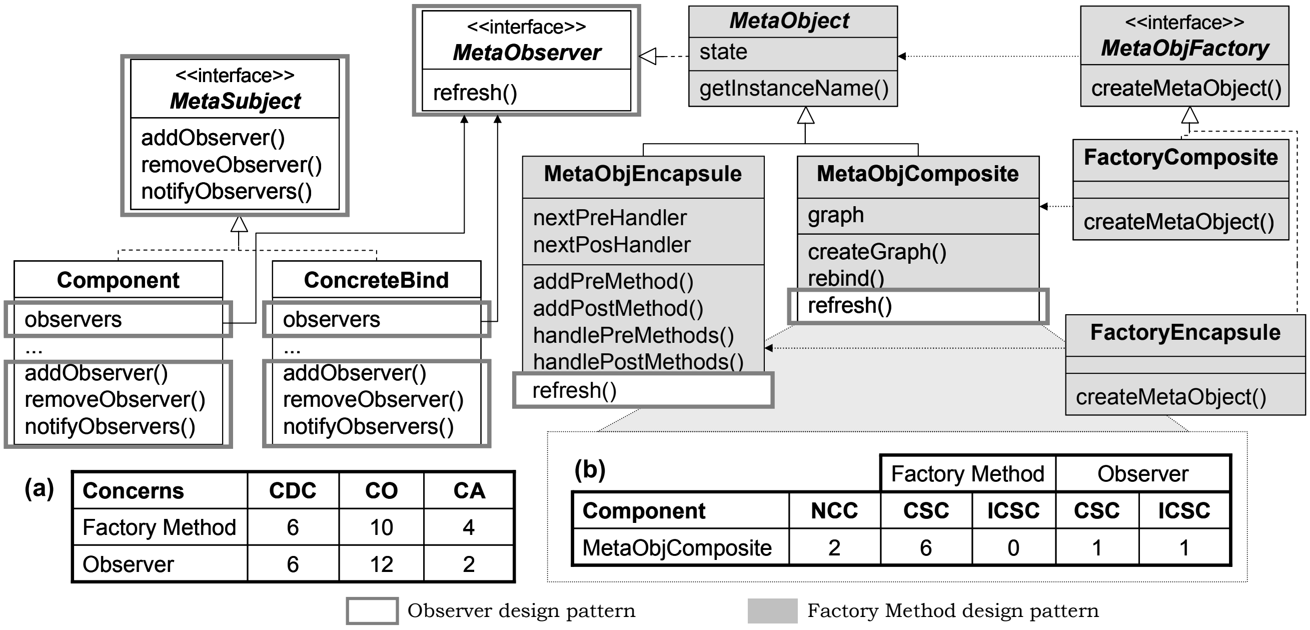
Although many design modularity flaws are related to the inadequate modularisation of concerns [2, 12, 14], most of the current quantitative assessment approaches do not explicitly consider concern as a measurement abstraction. This imposes certain shortcomings in order to effectively detect and correct design impairments. Also, this limitation becomes more apparent in the age of AOSD since different forms of design composition and decompositions have been brought.

To illustrate the limitations of conventional metric-based heuristic rules, we analyse the effectiveness of one of Marinescu's rules [18] in the light of a partial design showed in Figure 1. The analysed rule aims at detecting a specific kind of modularity flaws, 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 indicatives 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 symptom of design flaws caused by a poor modularisation of concerns [19]. Therefore, it might be avoided with the use of aspects. Marinescu's heuristic rule [18] for detecting Shotgun Surgery is based on two conventional coupling metrics. This rule is defined as follows.

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

CM stands for the Changing Method metric [17], 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], 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 be the 20% with highest CM and should not have CC higher than 5.

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, a rename of the `addObserver()` method in the `MetaSubject` interface causes updates to the classes `Component` and



(a)

| Concerns | CDC | CO | CA |
|---|---|---|---|
| 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

ConcreteBind (Figure 1) and several other classes which call addObserver().

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

In order to address the limitations of conventional heuristic rules discussed in the previous section, we defined a suite of heuristics that makes the evaluation process sensitive to the design concerns. The concern-sensitive heuristics are based on the combination of concern metrics and conventional metrics. This section defines the concern metrics used by the proposed suite of heuristics (Section 4). All the metrics presented in this section have a common underlying characteristic that distinguishes them from conventional metrics (Section 2): 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).

**Metrics Definition.** Each concern-driven metric of this section is presented in terms of a definition, 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. This figure shows a partial class diagram realising both Factory Method and Observer patterns [11]. Elements of the design are shadowed to indicate which pattern they implement. The concern metrics are computed based on the mapping of concerns to design elements. We used FEAT [21] for supporting semi-automatic concerns identification and ConcernMorph [8] 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 the 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 *Concern Attributes* (CA) counts the number of attributes (including inter-type attributes in aspects) that contribute to the realisation of a certain concern. Similarly, *Concern Operations* (CO) counts the number of methods, constructors, and advices that participate in the concern realisation. The goal of CA and CO 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 CO 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 (CA = 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 as mechanisms for supporting concern-sensitive modularity analysis. These rules are defined in terms of combined information collected from concern metrics (Section 3) and conventional modularity metrics. Table I presents four conventional metrics which are used by the proposed heuristics. Table I also includes references (1st column) and a short description of each metric (2nd column). 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 tasted 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, finding 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. If a concern is scattered, it is also tangled as a consequence. 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 denoting transitions between two concern classifications. Figure 2 also presents a diagram that 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 a further concern scattered analysis (by

TABLE I. Traditional software engineering 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 and advice of each class or aspect. |
| Coupling between Components (CBC) [3, 4, 22] | Counts the number of other classes and aspects which a class or an aspect is coupled to. |

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).

**R01 -** *Isolated*:
**if** NCC = 1 **for every component with** 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** (CA / NOA ≥ 0.5) **and** (CO / NOO ≥ 0.5) **for every component**
**then** Little Scattered Concern is Well-Encapsulated

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

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

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

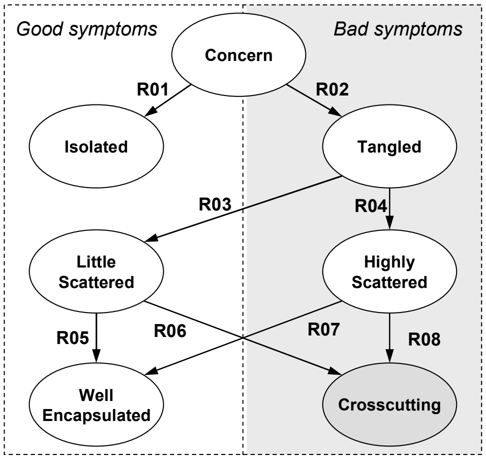


Figure 2. Concern classification and heuristics for concern diffusion

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. 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).

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 Concern Attributes (CA) and Concern Operations (CO) and two size metrics (Table I): 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 13 patterns of crosscutting concerns in our previous work [7]. This section illustrates how the concern-sensitive heuristics can be used to identify two of those crosscutting patterns, namely black sheep and octopus. Additional heuristics can be further defined to cover all 13 patterns. Black sheep and octopus were originally defined as follows.

*Black sheep is a property that crosscuts the system, but touches very few elements [5].*

*Octopus is a property which is well encapsulated in a few parts, but also spreads across other parts, as a crosscutting concern does [5].*

Therefore, black sheep and octopus are actually specialised categories of crosscutting concerns. Figure 3 shows two heuristic rules, R09 and R10, which aim at identifying black sheep and octopus concerns, respectively. Figure 3 also defines two conditions A (*Little Dedication*) and B (*High Dedication*) used by these rules. We explicitly separate the conditions from the rules not only to make the rules easier to understand but also to reuse the little dedication condition in both heuristics. In rules R09 and R10, a concern previously classified as crosscutting (Section 4.1) is thoroughly inspected in terms of how much each component dedicates itself to the crosscutting concern realisation.

```
Condition A - Little Dedication: (CA / NOA < 0.33) and (CO / NOO < 0.33)
Condition B - High Dedication: (CA / NOA ≥ 0.67) and (CO / 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 component and (High Dedication) for at least 1 component with Concern)
then Crosscutting Concern is Octopus
```

Figure 3. Heuristics for Black Sheep and Octopus

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).

The next rule R10 verifies if crosscutting concerns not classified as black sheep are a potential 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 it. Besides, 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].

### 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 three bad smells to illustrate the concern heuristics: Shotgun Surgery (Section 2.2), Feature Envy [10], and God Class [20]. 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]. God Class refers to a component (class or aspect) which performs most of the work, delegating only minor details to a set of trivial classes and using data from them [20]. We selected these three 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 Marinescu [18]. Hence, addressing different categories of flaws allows us to correlate our results with Marinescu's studies [18].

Figure 4 shows concern-sensitive heuristic rules for detecting the three bad smells aforementioned. The first rule, R11, 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 ((CA+CO) / (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.

The remaining two heuristics for bad smells R12 and R13 (Figure 4) are intended to detect Shotgun Surgery and God Class. Differently from the previous rules, R12 is composed of 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). Finally, a component is flagged as God Class (R13) if, besides 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 Evaluation

This section introduces a systematic evaluation of the concern-sensitive heuristics in 6 applications (Section 5.1) 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 three dimensions: (i) applicability for addressing metrics limitations (Section 5.2); (ii) accuracy to identify and classify crosscutting concerns in both OO and AO systems (Section 5.3); and (iii) the usefulness to detect design flaws in comparison with conventional heuristics (Section 5.4). Section 5.5 discusses threats to validity of our evaluation.

### 5.1 Selection of the Target Applications

In previous comparative studies [2, 9, 12-14], we applied AO metrics (e.g., metrics in Table I, Section 4) to a number of software systems. In this paper we selected six systems of our previous studies in order to apply and assess the accuracy of our concern-sensitive heuristics. Three of them [2, 13, 15] are medium-sized academic prototypes carefully designed with modularity attributes as main drivers. The other three systems are larger software projects, namely Health Watcher [14], CVS plugin [9], and a measurement tool [6]. Table II summarises the six 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. It allowed us to evaluate whether concern-sensitive heuristics are helpful to enhance a design assessment exclusively based on metrics (Section 5.2). 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 is the heterogeneity of concerns found in these systems (2nd column, Table II), 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), and multi-agent systems (6th). 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 these previous studies. Complete descriptions of such concerns and systems can be found in their respective publications (1st column of Table II).

### 5.2 *Solving Measurement Shortcomings*

The goal of this section is to discuss whether and how concern-sensitive heuristics address the limitation of conventional modularity assessment. For achieve this goal, this section presents 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. Due to space constraints, results of the conventional and concern metrics for all systems are available in our supplementary website [1].

Table II provides an overview of the application of the first two suites of rules (Sections 4.1 and 4.2) in the target systems. Note that the first study is subdivided in terms of design patterns (2nd column); the pattern roles were assessed (3rd column). The 3rd and 4th columns show all 23 concerns analysed in the respective studies and the best means of modularisation (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 systems. Their opinions were acquired by asking them to fill a questionnaire (available at [1]). The columns of Table II, labelled 01 to 10, present the answers 'yes' (y) or 'no' (n) of the concern diffusion heuristics (rules 01-08), black sheep (rule 09), and octopus (rule 10). Blank cells mean that the rule is not applicable. Finally, the last column indicates: (i) how the heuristics classify each concern and (ii) if the classification matches with the best proposed solution ('hit') or not ('fail'). The obtained heuristic classification of concerns includes isolated, well-encapsulated ('well-enc.'), crosscutting, black sheep, and octopus.

**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 II).

**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 in the light of a partial class diagram

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

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

**R11 -** *Feature Envy*:
**if** (*High Inter-Component Coupling*) **and** (*Low Intra-Component Coupling*) **and** (NCC > 1)
**then** Crosscutting Concern is Feature Envy

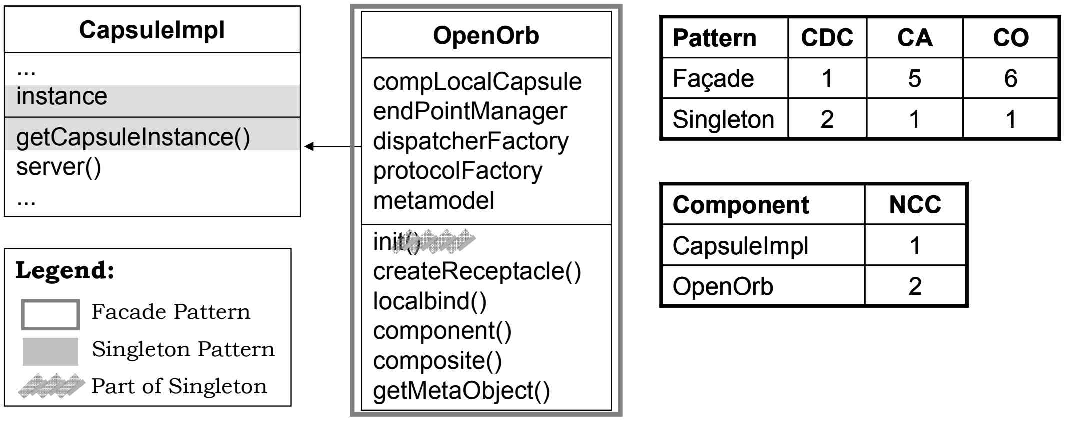
**R12 -** *Shotgun Surgery*:
**if** Concern is (*Tangled*) **and** (*Highly Scattered*) **and** (*Crosscutting*) **then** Crosscutting Concern is Shotgun Surgery

**R13 -** *God Class*:
**if** (NCC >2) **and** (NOA > (NOAsystem/NC)) **and** (NOO > (NOOsystem/NC)) **then** Component is a God Class

Figure 4. Heuristic rules for bad smells

TABLE II. RESULTS OF THE HEURISTICS FOR CONCERN DIFFUSION IN THE OO SYSTEMS.

| Studies | Nature of the Concerns | | Concerns | Best Solution | Heuristic Rules | | | | | | | | | | Results |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 10 | |
| (1) Patterns library [11, 14] | Design Patterns | Builder | Director role | OO | n | y | y | n | y | n | | | | | well-enc. / hit |
| | | Chain of Resp. | Handler role | AO | n | y | n | y | | | y | n | | | well-enc. / FAIL |
| | | Factory Method | Creator role | OO | n | y | n | y | | | n | y | n | n | crosscutting / FAIL |
| | | Observer | Observer role | AO | n | y | n | y | | | n | y | n | y | octopus / hit |
| | | | Subject role | AO | n | y | n | y | | | n | y | n | y | octopus / hit |
| | | Mediator | Colleague role | AO | n | y | n | y | | | n | y | n | y | octopus / hit |
| | | | Mediator role | AO | n | y | n | y | | | y | n | | | well-enc. / FAIL |
| (2) OpenOrb Middleware [2] | Design patterns and their compositions. | | Fact. Method | OO | n | y | y | n | y | n | | | | | well-enc. / hit |
| | | | Observer | AO | n | y | y | n | n | y | | | n | y | octopus / hit |
| | | | Façade | OO | y | n | | | | | | | | | isolated / hit |
| | | | Singleton | AO | n | y | y | n | n | y | | | y | n | black sheep / hit |
| (3) Measurement tool [2, 6] | | | Prototype | AO | n | y | y | n | n | y | | | y | n | black sheep / hit |
| | | | State | AO | n | y | y | n | y | n | | | | | well-enc. / FAIL |
| | | | Interpreter | AO | n | y | y | n | n | y | | | n | y | octopus / hit |
| | | | Proxy | AO | n | y | y | n | n | y | | | n | n | crosscutting / hit |
| (4) CVS core plugin [8] | Recurring architectural crosscutting and non-crosscutting concerns | | Exc. Handling | AO | n | y | n | y | | | n | y | n | y | octopus / hit |
| | | | Business | OO | n | y | n | y | | | y | n | | | well-enc. / hit |
| (5) Health Watcher [8, 13] | | | Concurrency | AO | n | y | n | y | | | n | y | n | y | octopus / hit |
| | | | Distribution | AO | n | y | n | y | | | n | y | n | y | octopus / hit |
| | | | Persistence | AO | n | y | n | y | | | n | y | n | y | octopus / hit |
| (6) Portalware [12] | Domain-specific concerns | | Adaptation | AO | n | y | y | n | n | y | | | n | n | crosscutting / hit |
| | | | Autonomy | AO | n | y | y | n | n | y | | | n | n | crosscutting / hit |
| | | | Collaboration | AO | n | y | n | y | | | n | y | n | y | octopus / hit |



| Pattern | CDC | CA | CO |
|---|---|---|---|
| Façade | 1 | 5 | 6 |
| Singleton | 2 | 1 | 1 |

| Component | NCC |
|---|---|
| CapsuleImpl | 1 |
| OpenOrb | 2 |

Figure 5. Concern metrics for Facade and Singleton.

presented in Figure 5. This figure highlights elements that implement the Singleton and Facade 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 would probably 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.

**Lack of mapping to flaw-causing concerns.** Two examples of problems are classified in this category: (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 (Table II) which is high scattered and tangled, e.g., CDC metric is 15 [1]. Nevertheless, in a more detailed analysis we found out that only six components have tangling among Collaboration and other system concerns. 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 II). Furthermore, Rule 10 identified that the octopus' tentacles only touch six components (which are indeed crosscut by 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 studies, 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., CA=10 and CO=22) [1]. Hence, the concern metrics have contradictory values in the sense that it is hard to conclude whether Adaptation is a crosscutting concern. Concern heuristics addressed this category of shortcomings in a number of cases. For instance, although Adaptation has contradictory results for concern metrics, the heuristics have successful identified it as an octopus (Table II), meaning that it is also a crosscutting concern.

### 5.3 *Accuracy of the Concern-Sensitive Heuristics*

This section evaluates the heuristics accuracy comparing their outcomes with the specialists' opinion or with the best known solutions (4th column of Table II). For this analysis, we count 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. In this step of our evaluation, in addition to the original designs, we also discuss the results for the aspectual ones (their detailed data for the heuristics application are available in our website [1]). The aspectual implementations aim at modularising one or more crosscutting concerns.

Table III provides overviews of the hits, false positives (FP) and false negatives (FN) of the rules for the 46 concern instances involved in this experiment (23 in OO and 23 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 III, the heuristics failed in about 20% of the case studies (6 false positives and 3 false negatives). Focusing on the OO designs, we found out 1 false positive and 3 false negatives. The false positive occurs with the Creator role of the Factory Method pattern (Table II). 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 III. STATISTICS FOR CONCERN DIFFUSION HEURISTICS.

| Studies | Hits (%) | FP (%) | FN (%) | Total (%) |
|---|---|---|---|---|
| OO | 19 (82.6) | 1 (4.3) | 3 (13.0) | 23 (100) |
| AO | 18 (78.3) | 5 (21.7) | 0 (0.0) | 23 (100) |
| OO + AO | 37 (80.4) | 6 (13.0) | 3 (6.5) | 46 (100) |

**Simple Program Slices.** The heuristics also presented 3 occurrences 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 II).

**Selection and Granularity of Concerns.** Moreover, the State 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 case study. 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 III presents five false positives 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 (Observer pattern) and Mediator and Colleague roles (Mediator 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.

**Aspect Precedence**. The Autonomy concern is also misclassified as crosscutting in the AO design of Portalware 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].

### 5.4 *Specific Design Flaws Detection*

This section evaluates whether our concern heuristics are useful to detect specific design flaws. In particular, we applied rules R11 to R13 (Figure 4) to identify Shotgun Surgery [10], Feature Envy [10] and God Class [20] bad smells in our six systems (Section 5.1). We also independently applied the 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) by undertaking a manual investigation in all suspect design fragments. In circumstances when the existence of a bad smell was not clear about, we contacted professionals and researchers with long-term experience on the development and assessment of the target designs. The manual investigation allowed us to verify whether the suspect design fragments were indeed affected by the design flaw.

After this inspection, the total number of suspects for each bad smell was classified in 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 IV 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 IV 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 and conventional rules presents the results in terms of number of classes or operations.

Data of Table IV show that concern-sensitive heuristics presented superior accuracy than conventional rules for detecting all three studied design flaws. The former presented no more than 20% of false positives, whereas the latter exhibited many false positives. 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.

TABLE IV. STATISTICS FOR THE HEURISTICS RELATED TO BAD SMELLS.

| Bad Smell | Concern-Sensitive Rules | | Conventional Rules | |
|---|---|---|---|---|
| | Hits (%) | FP (%) | Hits (%) | FP (%) |
| Shotgun Surgery | 8 (89%) | 1 (11%) | 9 (56%) | 7 (44%) |
| Feature Envy | 3 (100%) | 0 (0%) | 1 (17%) | 5 (83%) |
| God Class | 8 (80%) | 2 (20%) | 1 (17%) | 5 (83%) |

### 5.5 *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, detecting the Exception Handling concern 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 evaluated the applicability of the approach by defining thirteen concern-sensitive heuristic rules that are classified in three categories. Based on their application to six 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 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 5.1).

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.

## 6 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 two 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 for determining concerns that should be aspectised. It also pointed out that the heuristics have around 80% of precision (Sections 5.3 and 5.4). Additionally, heuristics provided enhancements both in purely OO and in aspectual design assessment. But, we also identified some problems in the accuracy of the heuristics when (Section 5.3): (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 (Tables II and III). Of course, our study is an initial stepping stone on understanding how concerns can be useful assessment abstractions. Further analyses are required to confirm or refute our findings.

## ACKNOWLEDGMENT

This work was partially supported by FAPESB and by the National Institute of Science and Technology for Software Engineering (INES), funded by CNPq (grant 573964/2008-4). We would like to thanks the anonymous reviewers for their valuable comments. Eduardo was supported by CAPES.

## REFERENCES

[1] Figueiredo, E., Sant'Anna, C., Garcia, A. and Lucena, C. Concern-Sensitive Design Heuristics. URL: http://www.lancs.ac.uk/postgrad/figueire/concern/heuristics/

[2] Cacho, N. et al. (2006) Composing Design Patterns: A Scalability Study of Aspect-Oriented Programming. *Proc. of the Int'l Conf. on Aspect-Oriented Software Dev. (AOSD)*, Germany.

[3] Ceccato, M., and Tonella, P. (2004) Measuring the Effects of Software Aspectization. *Proc. of the 1st workshop on Aspect Reverse Engineering*, The Netherlands.

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

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

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

[7] Figueiredo, E. et al. (2009) Crosscutting Patterns and Design Stability: An Exploratory Analysis. *Proc. of the Int'l Conf. on Program Comprehension (ICPC)*, Vancouver, Canada.

[8] Figueiredo, E., Whittle, J. and Garcia, A. (2009) ConcernMorph: Metrics-based Detection of Crosscutting Patterns. *Proc. of the joint ESEC/FSE meeting*, demo session, Amsterdam.

[9] Filho, F. et al. (2006) Exceptions and Aspects: The Devil is in the Details. *Proc. of Int'l 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, 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.

[14] Greenwood, P. et al. (2007) On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study. *Proc. of ECOOP*, Berlin, Germany.

[15] Hannemann, J., and Kiczales, G. (2002) Design Pattern Implementation in Java and AspectJ. *Proc. of the Int'l Conf. on OOP, Systems, Lang. and App. (OOPSLA)*, 161-173.

[16] Kiczales, G. et al. (1997) Aspect-Oriented Programming. *In Proc. of the European Conf. on Object-Oriented Programming (ECOOP)*, LNCS 1241, Springer, 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. *Proc. of the International Conference on Software Maintenance (ICSM)*, 350 - 359.

[19] Monteiro, M., and Fernandes, J. (2005) Towards a Catalog of Aspect-Oriented Refactorings. *Proc. of the Int'l Conf. on AOSD*, Chicago, 111-122.

[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. *Proc. of the Braz. Symp. on Software Eng. (SBES)*, 19-34.

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