Crosscutting Patterns and Design Stability: An Exploratory Analysis

Eduardo Figueiredo[1], Bruno Silva[2], Claudio Sant'Anna[3],
Alessandro Garcia[4], Jon Whittle[1], Daltro Nunes[2]

[1]Computing Department, Lancaster University, United Kingdom
[2]Institute of Informatics, Federal University of Rio Grande do Sul, Brazil
[3]Computer Science Department, Federal University of Bahia (UFBA), Brazil
[4]Software Engineering Laboratory, Catholic University of Rio de Janeiro, Brazil

{figueire, whittle}@comp.lancs.ac.uk, {bruno.carreiro, daltro}@inf.ufrgs.br,
santanna@dcc.ufba.br, afgarcia@inf.puc-rio.br

Abstract

It is often claimed that inaccurate modularisation of crosscutting concerns hinders program comprehension and, as a consequence, leads to harmful software instabilities. However, recent studies have pointed out that crosscutting concerns are not always harmful to design stability. Hence, software maintainers would benefit from well documented patterns of crosscutting concerns and a better understanding about their actual impact on design stability. This paper presents a catalogue of crosscutting concern patterns recurrently observed in software systems. These patterns are described and classified based on an intuitive vocabulary that facilitates their recognition by software engineers. We analysed instances of the crosscutting patterns in object-oriented and aspect-oriented versions of three evolving programs. The outcomes of our exploratory evaluation indicated that: (i) a certain category of crosscutting patterns seems to be good indicator of harmful instabilities, and (ii) aspect-oriented solutions were unable to modularise concerns matching some crosscutting patterns.

1. Introduction

One of the key benefits of separation of concerns is that it supports developers and maintainers to gradually build up an understanding of the software system [14]. However, recent empirical studies [4, 5, 7, 10] have pointed out that concerns cutting across the software modular structure – hereafter called *crosscutting concerns* – hinder program comprehension. A concern is any critical or important consideration to one or more stakeholders involved in the software specification, implementation or evolution [21].

It has been observed that the inappropriate modularisation of crosscutting concerns, such as error handling and distribution, leads to undesirable software instabilities [5, 21, 23]. A module (or component) is considered to be harmfully instable when its modularity flaws – e.g., high coupling and low cohesion – or observed faults start to increase through the software history [13]. Concern scattering tends to both destabilise software modularity properties and facilitate the introduction of faults according to empirical assessments [5, 7, 10]. There is also initial evidence that not all types of crosscutting concerns are harmful to a software system [7, 18].

However, there is not much understanding on what specific types of crosscutting concerns are likely to deteriorate design stability. This knowledge is particularly important with the advent of advanced modularity techniques, such as aspect-oriented programming (AOP) [14]. Stability analysis of crosscutting concerns can not be carried out without their systematic classification. The realisation of crosscutting concerns affects modularity units of a system in significantly-different ways, ranging from a few elements in a specific inheritance tree to code fragments scattered across the entire system.

Recent preliminary work on concern-based analysis [4, 6, 18] focused on investigating how crosscutting concerns affect program comprehensibility and maintainability. There also exists a number of complementary tools for concern mining [1], concern measurement [5, 21], and refactoring of crosscutting concerns [11, 17]. However, such analyses and tools suffer from the lack of a unified catalogue documenting recurring forms of crosscutting concerns.

In this context, this paper presents a systematic documentation of typical types of crosscutting concerns (*crosscutting patterns*). We rely on set theory

and on a generic meta-model (Section 2) for characterising the notion of crosscutting concerns in software systems. Based on this formalism, we provide a comprehensive categorisation for the crosscutting patterns which facilitates their understanding and investigation (Section 3). The classification was based on the different characteristics of the concern realisations, which are likely to have different impacts on software stability. As the formalisation is agnostic to language intricacies, the proposed crosscutting patterns can be used to anticipate instabilities in both implementation-level and design-level artefacts.

An exploratory evaluation of our catalogue of crosscutting patterns is performed using three heterogeneous applications (Section 4). Our key findings suggest that crosscutting concerns following some specific patterns may not be easily modularised, even with the use of AOP mechanisms. Furthermore, we perform an exploratory analysis on the correlation of design stability and crosscutting concern patterns. Finally, we discuss some related work (Section 5) before concluding this paper (Section 6).

2. Analysing crosscutting concerns

Section 2.1 introduces the formalism used in an analysis of crosscutting concerns. Based on this formalism, Section 2.2 motivates this paper by illustrating how recurring patterns of crosscutting concerns affect quality properties of a design.

2.1. Basic definitions

This section defines a concern-based meta-model and formalism based on set theory. We seek to use terminology and notation that are, as much as possible, agnostic to language and paradigm peculiarities. Figure 1 presents the simplified concern meta-model for a generic system representation. It not only defines possible relations of concerns and the system's structure, but also subsumes key abstractions for module specifications. Each type of abstraction is called an *element*. Concerns can be realized by an arbitrary set of elements. A system *S* consists of a set of components, denoted by *C(S)*. The notion of components expresses either a class or an interface in an OO language, an aspect in an AO language [14], or even a component in an architecture description language [22]. Each component *c* consists of a set of attributes, *Att(c)*, a set of operations, *Op(c)*, and a set of declarations, *Dec(c)*. The set of members of a component *c* is defined by $M(c) = Att(c) \cup Op(c) \cup Dec(c)$. Our meta-model can be extended for further concern-based analysis purposes.

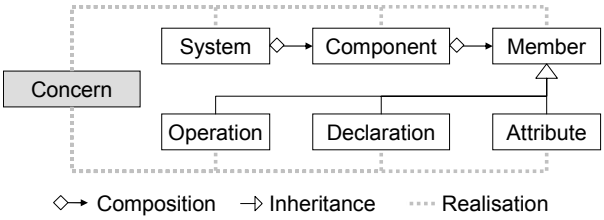

Figure 1. Simplified meta-model of concern realisation

According to our meta-model, a concern is an abstraction addressed by structural elements which realise it. Examples of concerns are software requirements, design patterns [9], and product-line features [7]. To identify crosscutting patterns, it is first necessary to assign each concern of interest to structural elements that realise it [6, 21]. The set of concerns addressed by the system *S* is defined as *Con(S)*. A concern *con* can be realised by a set of components, *C(con)*, a set of attributes, *Att(con)*, a set of operations, *Op(con)*, and/or a set of declarations, *Dec(con)*. The set of members that implement a concern *con* is defined as $M(con) = Op(con) \cup Att(con) \cup Dec(con)$. This formalism is used by all crosscutting patterns presented in this paper, but further definitions are presented for each crosscutting pattern's expression as necessary.

2.2. An illustrative model instantiation

The aforementioned formalism is abstract enough to be instantiated for different modelling and programming languages. Table 1 demonstrates how our generic meta-model (Figure 1) is instantiated for supporting concern-oriented analysis of an object-oriented (OO) design. For example, system represents the OO model while a component can be either a class or an interface. The declaration abstraction is not implemented by any element of this language.

Table 1. Meta-model instantiation for OO design.

Element	Instantiation
System	OO Design Model
Concern	Concern
Component	Class and Interface

Element	Instantiation
Attribute	Class Variable and Field
Operation	Method and Constructor
Declaration	-

Once the meta-model is instantiated, concern-oriented analysis can be performed using the proposed formalism. For example, Figure 2 shows a partial design of a Web-based system, called Health Watcher. The figure highlights elements of the design realising the Observer and Singleton design patterns [9]. Those two design patterns are the concerns of interest in this

case. It is observed in this figure how concerns manifest themselves in distinct ways over the design model. For instance, the Observer pattern is partially modularised by two interfaces, but crosscuts four other classes. Singleton, on the other hand, only affects a small portion of one class in this partial model.

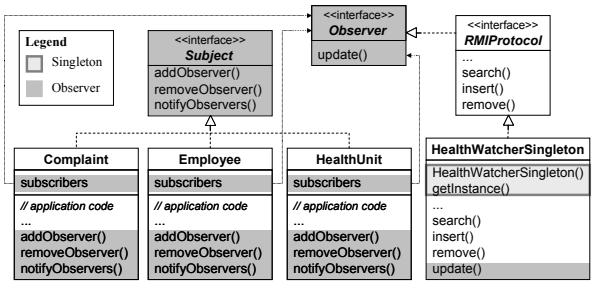

Figure 2. Examples of crosscutting design patterns

The different ways that concerns crosscut a system are associated with maintainability properties of the system, such as design stability and error proneness. For example, a recent work [5] has shown that the more scattered a concern is the more faults it causes in components realising it. However, concern scattering by itself is not a good indicator of design quality. For example, although the `Observer` and `Subject` interfaces (Figure 2) realise a single concern, they might be sources of design instability. That is, changing a method signature of these interfaces would trigger many other updates to elements realising the Observer concern. A rename of the `addObserver()` method in the `Subject` interface, for instance, causes updates to the classes `Complaint`, `Employee` and `HealthUnit` and potentially to several other classes calling `addObserver()`. In the following section we take many properties of crosscutting concerns into consideration to formalise 13 crosscutting patterns. In Section 4 we correlate each of those crosscutting patterns to design stability.

3. Patterns of crosscutting concerns

This section presents a catalogue of crosscutting concern patterns (or *crosscutting patterns*) showing recurrent ways in which concerns manifest themselves in a system. We present each crosscutting pattern in terms of a textual description, a formal definition using our formalism (Section 2.1), and an example. As far as the examples are concerned, we use an abstract pictorial representation and may also refer to our case studies (Section 5). The crosscutting patterns are divided into four categories as presented in the following subsections.

3.1. Flat crosscutting shapes

This section describes and formalises three patterns of crosscutting concerns, *Black Sheep*, *Octopus*, and *God Concern*. The first two patterns (Octopus and Black Sheep) have already been identified in previous work [4]. God Concern and the other 10 crosscutting patterns of this paper represent our original contribution towards a unified catalogue.

Black Sheep. The *Black Sheep* crosscutting pattern (also called *Lazy Concern*) is defined as a concern that crosscuts the system but touches very few elements in distinct places [4]. For example, Black Sheep occurs when only a few scattered methods and attributes are required to realise a concern in the design artefacts. Also, there is not a single component whose main purpose is to implement this concern. Figure 3 (left) presents an abstract representation of a Black Sheep concern. The shadow grey areas in this figure indicate elements of the components (represented by boxes) realising the concern under consideration. Taking the first four boxes into account, the Black Sheep instance is realised by few elements of two components.

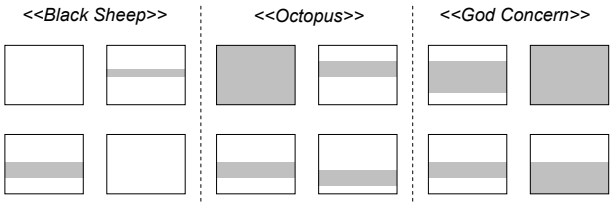

Figure 3. Patterns of flat crosscutting shapes

For a formal definition of the Black Sheep crosscutting pattern, let's consider *c* a component in *C(S)* and *con* a concern in *Con(S)*. Also, *dedication(c, con)* indicates how much of the component *c* is used to implement the concern *con*. Then, we define a concern *con* to be Black Sheep if and only if (iff):

$$\text{BlackSheep}(con) \Leftrightarrow \forall\, c \in C(con)\ \text{dedication}(c, con) < 33\ \%$$

This expression says that all components realising a Black Sheep concern dedicate less than 33 % of their data and behaviour to this concern. The choice of 33% is derived from general evidence-based guidelines for selecting thresholds [16], and it could be obviously adapted according to application-specific contexts and the designer's experience[1]. An example of Black Sheep is the Singleton design pattern [9] presented in the partial design of Figure 2. This Singleton solution matches the Black Sheep pattern because the

[1] The same rationale applies to the other thresholds in this paper.

`HealthWatcherSingleton` class requires only two operations, the `getInstance()` method and one constructor, to realise the pattern's concern.

Octopus. The second crosscutting pattern in this group, *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 [4]. Figure 3 (middle) presents an illustrative example of the Octopus crosscutting pattern. Note that one component is completely dedicated to the concern realisation (first box that represents the *octopus' body*), while three other components (representing *tentacles*) dedicate only small parts to the same concern. We use the previously defined relation *dedication(c, con)* to give the following formal definition of Octopus. As stated in the expression below, a concern is Octopus iff at least one component dedicates less than 33% and another dedicates more than 67% to the concern realisation.

$$Octopus(con) \Leftrightarrow \exists\ c\ \exists\ d \in C(con)\ dedication(c, con) < 33\ \% \wedge dedication(d, con) > 67\ \%$$

Figure 2 in Section 2.2 presents an instance of the Observer design pattern [9] which matches our definition of Octopus. In other words, it has two interfaces `Subject` and `Observer` completely dedicated to the concern implementation (the octopus' body) and four classes with only few methods and attributes realising Observer (octopus' tentacles).

God Concern. The *God Concern* crosscutting pattern occurs when a concern encompasses too much scattered functionality of the system. In other words, the elements realising this concern address too much or do too much across the software system. The basic idea behind modular programming is that a broadly-scoped concern should be analysed as smaller "modular" sub-parts (divide and conquer [15]). However, God Concern instances do not adhere to this principle hindering concern-based analysis.

Figure 3 (right) 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. However, at least one component affected by a God Concern instance does not have the main purpose of realising it (this is a guarantee that this concern is crosscutting). The formalisation of God Concern considers the overall dedication of the system's components to a given concern. Hence, it requires our *dedication()* relation to be overridden as given below. The average dedication of all components in *C(S)* realising God Concern is higher than 33%. In other words, at least 33% of the system overall functionality is assigned to this concern.

$$GodConcern(con) \Leftrightarrow \exists\ c \in C(con)\ dedication(c, con) < 50\% \wedge dedication(C(S), con) > 33\%$$

Where,

$$dedication(C(S), con) = \frac{\sum^{c \in C(S)} dedication(c, con)}{|\ C(S)\ |}$$

An example of God Concern can be found in Health Watcher [10]. The Business concern (an architectural layer) is considered as an architecturally-relevant concern of major interest in this application and, as such, its sub-parts should be also modularised in later design artefacts. This concern encompasses different sub-concerns, such as Complaint, Disease, Employee, and Health Unit (among others). Although all these concerns are related to the business logic, a concern-based analysis should consider its several sub-concerns so that each of them represents a different modular slice of the Business rules.

3.2. Inheritance-wise concerns

This group includes two crosscutting patterns identified in inheritance trees.

Climbing Plant. *Climbing Plant* is a pattern of crosscutting concerns that affect the root of an inheritance tree and propagate their structure to all children in the tree. This crosscutting structure can be totally or partially propagated to the root descendents. However, all descendents (also called *branches*) of the concern root are somehow affected. The main problem caused by concerns matching this crosscutting pattern is related to implicit dependencies involving their forming components. These dependencies become clear when a change targeting a concern branch ripples through the root to other branches.

Hereditary Disease. As Climbing Plant, *Hereditary Disease* affects the root of an inheritance tree and, then, propagates its crosscutting structure to *some* root descendents. However, a concern matching the Hereditary Disease crosscutting pattern does not manifest itself in all nodes of a tree, i.e., some nodes are *disease-free*. Figure 4 illustrates the key difference between these two crosscutting patterns. Note that there is a disease-free node in the Hereditary Disease example which is not allowed in the Climbing Plant pattern. Apart from that, the modularity problems are similar and require analogous solutions. Recurring instances of these two crosscutting patterns appear in all three considered case studies (Section 5). For example, the Hereditary Disease crosscutting pattern can be noticed in the design presented in Figure 2.

An additional transitive relation, *descendent(c1, c2)*, is required to formalise Climbing Plant and Hereditary Disease. This relation returns true if *c1* ∈ *C(S)* directly or indirectly inherits from *c2* ∈ *C(S)*. Note that *Branches* and *DiseaseFreeNodes* are sets of components as formalised below.

$$ClimbingPlant(con) \Leftrightarrow Branches(con) \neq \emptyset \wedge DiseaseFreeNodes(con) = \emptyset$$

$$HereditaryDisease(con) \Leftrightarrow Branches(con) \neq \emptyset \wedge DiseaseFreeNodes(con) \neq \emptyset$$

Where,

$root := r \in C(con) \mid \forall\, c \in Branches(con)\ descendent(c, r) \wedge \nexists\, d \in C(con)\ descendent(r, d)$

$Branches(con) := \{ c \in C(con) \mid descendent(c, root) \}$

$DiseaseFreeNodes(con) := \{ c \in C(S) \mid descendent(c, root) \wedge c \notin C(con) \}$

3.3. Communicative concerns

This section presents four patterns of crosscutting concerns: *Tree Root*, *Tsunami*, *King Snake*, and *Neural Network*. The particular characteristic of components realising concerns of this pattern category is that they have coupling connections among them. The different ways in which components connect to each other define the distinct crosscutting patterns.

Tree Root. The *Tree Root* pattern is formed by two kinds of components: *trunk* and *feeders*. The trunk of Tree Root is a component that only receives incoming connections from other components. The feeders, on the other hand, are components which directly or indirectly connect to the trunk. An abstract representation of Tree Root is presented in Figure 4. This figure shows three feeders connected to a trunk.

For a formal definition of Tree Root, we define the *connected(c1, c2)* relation that maps to a Boolean value indicating if a component *c1* connects to a component *c2* (both *c1* and *c2* realising the concern *con*). This relation is transitive, i.e., given *c1*, *c2* and *c3* ∈ *C(con);* if *connected(c1, c2)* and *connected(c2, c3)* then *connected(c1, c3)*. We also define the *trunk* and the *Feeders* set. According to these expressions, a trunk is a component realising a concern *con* and there is no other component in *C(con)* that connects to it. Similarly, *Feeders(con)* is a subset of *C(con)* so that each feeder component is connected to the trunk. We define a concern to be Tree Root iff there is at least one component in the Feeders set.

$$TreeRoot(con) \Leftrightarrow Feeders(con) \neq \emptyset$$

Where,

$trunk := t \in C(con) \mid \nexists\, d \in C(con)\ connected(t, d)$

$Feeders(con) := \{c \in C(con) \mid connected(c, trunk)\}$

Tsunami. The *Tsunami* crosscutting pattern is formed by a core component, named *wave source*, which directly or indirectly connects to other components realising the same concern. This pattern resembles wave propagation through coupling connections. Unlike Tree Root, connections do not converge to a single component in the Tsunami pattern. In fact, Tsunami has the inverse configuration in comparison to Tree Root. The extremes in the "channel" of called components are named *wave limits* (i.e., those realising the concern that just receive connections).

Figure 4 shows the overall Tsunami structure. This figure also provides us with a comparison of the Tree Root and Tsunami crosscutting patterns. In particular, note the directions of connections, i.e., converging to a trunk in the former and diverging from a wave source in the latter. The Tsunami formal definition (below) is also similar to the Tree Root one. It states that a wave source is a component realising a concern *con* in *C(con)* and that Waves is a subset of components in *C(con)*. Furthermore, the wave source connects to every element of the Waves set.

$$Tsunami(con) \Leftrightarrow Waves(con) \neq \emptyset$$

Where,

$source := s \in C(con) \mid \nexists\, d \in C(con)\ connected(d, s)$

$Waves(con) := \{ c \in C(con) \mid connected(source, c) \}$

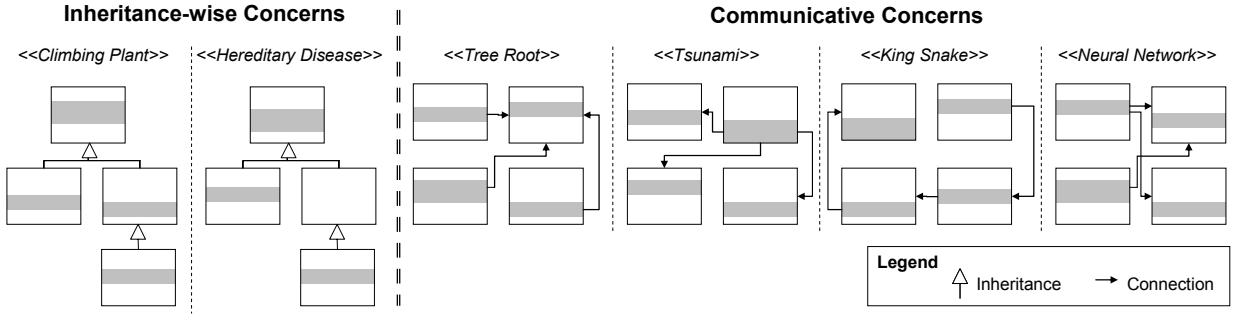

Figure 4. Abstract representation of inheritance-based and communicative concerns

King Snake. The third crosscutting pattern in this group, *King Snake*, is characterised by a large chain of connected components realising a given concern. The first component in the chain is named *head* of the snake and the final connected component (end of the chain) is named *tail*. In cases where a concern has more than one chain of components in its structure, we consider as King Snake the longest chain.

Figure 4 illustrates an example of King Snake. In that figure, four components are connected to one another due to the concern under consideration. To formalise King Snake, we define *Chain(con)* as a set of components realising a concern *con* and participating in a chain that connects them. We also use the previously defined *trunk* and *source* expressions.

$$KingSnake(con) \Leftrightarrow Chain(con) \neq \varnothing$$

Where,

$head := source$; $tail := trunk$

$$Chain(con) := \{ c \in C(con) \mid connected(head, c) \wedge connected(c, tail) \}$$

Neural Network. The last crosscutting pattern of this group, *Neural Network*, consists of a large net of inter-connected components. Each component in the net is connected to one or more other components as presented in the rightmost side of Figure 4. The overall configuration of this pattern resembles an artificial neural network including (i) components which are not called by any other (*input layer*); (ii) components which do not connect to any other (*output layer*); (iii) and optional components at the intermediary structure receiving and making connections (*hidden layers*). A concern is Neural Network iff there are connecting paths from the input to the output layer.

$$NeuralNetwork(con) \Leftrightarrow \forall\, i \in InputLayer(con)\ \exists\, o \in OutputLayer(con)\ connected(i, o)$$

Where,

$$InputLayer(con) := \{ i \in C(con) \mid \forall\, c \in C(con)\ connected(i, c) \wedge \nexists\, d \in C(con)\ connected(d, i) \}$$

$$OutputLayer(con) := \{ o \in C(con) \mid \forall\, c \in C(con)\ connected(c, o) \wedge \nexists\, d \in C(con)\ connected(o, d) \}$$

3.4. Other crosscutting patterns

This group includes four patterns of crosscutting concerns, *Copy Cat*, *Dolly Sheep*, *Data Concern*, and *Behavioural Concern*, which do not fit in the other categories. These crosscutting patterns express either replicated code or concerns lacking data or behaviour.

Copy Cat and Dolly Sheep. *Copy Cat* is a crosscutting pattern where replicated code implements the same concern. In other words, the given concern is implemented by similar pieces of code in different components. These duplications can occur, for example, as a result of copy and paste practices and they increase the overall costs of maintenance activities [8]. Copy Cat also occurs when pieces of code implementing such concern are almost identical, varying only in small details. In either similar or identical code, every time one piece of concern code is modified, other copies are likely to require similar modifications. A special type of Copy Cat occurs when the concern is also classified as Black Sheep. In this case, the crosscutting pattern is called *Dolly Sheep* since it represents a "replicated sheep".

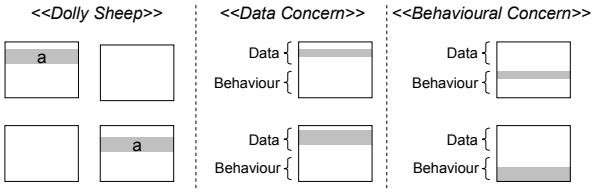

Figure 5. Other recurrent crosscutting patterns

Figure 5 presents an abstract representation of the Dolly Sheep crosscutting pattern (which is also Copy Cat). The label 'a' in different boxes of this figure indicates that the shadowed areas contain similar code. To formalise those two patterns of crosscutting concerns, we define *similar(m1, m2)*. This function verifies whether two members (*m1* and *m2*) of two different components have similar code. Then, Copy Cat and Dolly Sheep can be expressed as:

$$CopyCat(con) \Leftrightarrow \exists\, c\ \exists\, d \in C(con)\ \exists\, m \in (M(c) \cap M(con))\ \exists\, n \in (M(d) \cap M(con))\ similar(m, n)$$

$$DollySheep(con) \Leftrightarrow CopyCat(con) \wedge BlackSheep(con)$$

Data Concern and Behavioural Concern. The last two crosscutting patterns are Data Concern and Behavioural Concern. *Data Concern* (or *Brain*) is a pattern to name a crosscutting concern mainly composed of data, i.e., it has no (or very few) associated behaviour. In fact, existing behaviour in a Data Concern instance is due to data accessors (e.g., get and set methods). Differently from Data Concern, *Behavioural Concern* (or *Brainless*) is only composed of behaviour, i.e., it has no associated data.

The formal definitions of Data Concern and Behavioural Concern use two additional expressions: *data(m)* and *dataAccessor(m)*. These relations return a Boolean value indicating whether the member $m \in M(c)$, is data (e.g., field or class variable) or a data

accessor. Given those definitions, Data Concern and Behavioural Concern are formalised as follows.

$$DataConcern(con) \Leftrightarrow \forall\, c \in C(con)\ \forall\, m \in (M(c) \cap M(con))\ data(m) \vee dataAccessor(m)$$

$$BehaviouralConcern(con) \Leftrightarrow \forall\, c \in C(con) \nexists\, m \in (M(c) \cap M(con))\ data(m)$$

4. Evaluation and discussions

This section performs an exploratory analysis of the proposed catalogue of crosscutting patterns. First, Section 4.1 presents how we identify those patterns of crosscutting concerns in a set of target applications. Then, Section 4.2 discusses the most recurring patterns by analysing the number of pattern instances found in each application. Finally, Section 4.3 assesses the degree of correlation between crosscutting patterns and design stability in two evolving systems.

4.1. Detecting crosscutting patterns

Three applications were used in our exploratory evaluation. The identification of crosscutting patterns in these systems was particularly interesting as they are from significantly different application domains. The first one is a library of implemented design patterns (DPs) [9] developed by Hannemann and Kiczales [12]. The others are two medium-sized applications: a Web-based information system called Health Watcher (HW) [10, 22] and a MVC-based software product line for handling data on mobile devices, called MobileMedia (MM) [7]. Java and AspectJ implementations were available for all three applications. We considered 7 successive MM releases [7, 20] and 10 HW releases. Each release evolved from the previous one by adding new functionalities to its specific application or by refactoring its design to achieve a better modularised structure. Table 2 lists the analysed concerns for each target application. The selected concerns (and their representations in the target systems) came from previous studies [7, 10, 12, 22, 23], allowing us to compare our results to other experiments.

In order to detect patterns of crosscutting concerns in those applications, we used three conventional analysis techniques: detection strategies [19], tools for clone detection [1], and graph search algorithms [15]. In fact, we had to adapt these techniques to make them sensitive to the analysed concerns. For example, inspired by Marinescu's work [19], we defined specific *concern-based detection strategies* to identify some of our proposed crosscutting patterns. Marinescu's proposal is based on conventional metrics aiming at detecting traditional design flaws, such as bad smells [8]. To achieve our goal, the detection strategy mechanism is extended with recently proposed concern metrics [5, 6]. A detailed discussion of the used detection mechanism is out of the scope of this paper, but readers may want to cover this part by consulting our supplementary material [3].

Table 2. List of analysed concerns

Case Study	Concerns
Design Patterns (DPs)	44 design pattern roles: Factory and Product (Abstract Factory); Adapter, Adaptee, and Target (Adapter); Subject and Observer (Observer); Singleton; etc.
Mobile Media (MM)	13 product-line features: Capture, Controller, Copy, Exception Handling (EH), Favourites, Labelling, Media, Music, Persistence, Photo, SMS, Sorting, and Video.
Health Watcher (HW)	5 architectural concerns (Business, Concurrency, EH, Distribution, and Persistence) and 5 design patterns (Abstract Factory, Adapter, Command, Observer, State).

We detected crosscutting patterns in Java and AspectJ implementations of the target applications. Since the formalism adopted in Section 3 is abstract, it needs to be instantiated to particularities of each programming language. Table 3 presents a possible instantiation of the abstract elements of the proposed concern meta-model for both Java and AspectJ. It also describes the computation of some definitions given for the patterns' formalisation in the previous section. The definitions of Table 3 enable a precise detection of the crosscutting patterns in our case studies.

Table 3. Model instantiation for Java and AspectJ systems

Definition	Java Implementation	AspectJ Implementation
System	Java System	AspectJ System
Concern	Concern	Concern
Component	Class and Interface	Class, Interface, and Aspect
Attribute	Class Variable and Field	Class Variable, Field, and Inter-type Attribute
Operation	Method and Constructor	Method, Constructor, Advice, and Inter-type Operation
Declaration	-	Pointcut and Declare State-ment (e.g., Parents, Soft)
dedication(c, con) Section 3.1	Percentage of attributes and operations realising a concern (*con*) in each component (*c*)	
connected(c1, c2), Section 3.3	Explicit references (e.g., method calls and attribute accesses) from one component (*c1*) to another (*c2*)	
similar(m1, m2), Section 3.4	Manual inspection of the concern code to decide whether a member (*m1*) is similar to another (*m2*).	

4.2. Patterns in OO and AO systems

This section reports and discusses patterns of crosscutting concerns detected in the three analysed applications. Table 4 presents the number of crosscutting pattern's instances in each Java implementation (and multiple releases in the MM case)

considering all analysed concerns (Table 2). Rows of Table 4 list the 13 crosscutting patterns while columns show the systems. The last row indicates the number of concerns analysed in that specific application. The number of concerns varies through the MM releases because new concerns were introduced as the system evolved [7]. While space constraints prevent the inclusion of complete results (e.g., for all HW releases), all data per concern (and per component) are available from a supplementary website [3].

Table 4. Crosscutting patterns per system

Crosscutting Patterns	MobileMedia Releases							HW R9	DPs
	01	02	03	04	05	06	07		
Black Sheep (BS)	1	1	2	2	2	2	2	1	5
Octopus (Oct)	3	5	5	6	7	8	10	5	11
God Concern (GC)	1	1	1	1	1	0	0	2	10
Climbing Plant (CP)	4	4	4	7	8	9	10	6	25
Her. Disease (HD)	0	0	0	0	0	7	9	2	9
Tree Root (TR)	3	4	4	7	6	7	9	8	6
Tsunami (Tsu)	3	4	4	5	6	8	9	6	16
King Snake (KS)	3	3	4	8	7	8	11	4	8
Neur. Network (NN)	3	3	4	8	7	8	11	4	4
Copy Cat (CC)	2	2	2	2	2	3	3	6	11
Dolly Sheep (DS)	0	0	0	0	0	0	0	0	2
Data Concern (DC)	0	0	0	0	0	0	0	0	1
Beh. Concern (BC)	0	0	0	0	0	0	0	0	10
# of concerns	**5**	**6**	**7**	**8**	**9**	**11**	**13**	**10**	**44**

Table 4 shows that some crosscutting patterns are more common in particular cases. For example, the last three patterns (Dolly Sheep, Data Concern, and Behavioural Concern) could only be identified in the design pattern library. This result is somehow expected due to the interesting heterogeneous forms of concerns found in design patterns. Someone could alternatively argue that, based on our initial analysis, such crosscutting patterns seem to be specific to design pattern implementations. However, the Gang-of-Four design patterns and their variants [9] are consistently instantiated in all types of application domains and, therefore, should be carefully analysed.

Furthermore, Hereditary Disease instances only emerged in situations with frequent use of inheritance, such as HW, last two MM releases and some design patterns. This type of crosscutting pattern seems to be particularly interesting to analyse as they often manifest in program families, such as frameworks and product lines [7]. Program families rely extensively on the use of abstract classes and interfaces in order to implement variabilities. The inappropriate modularisation of such crosscutting concerns might lead to future instabilities in the design of the varying modules.

More interestingly, there are six crosscutting patterns (Octopus, Climbing Plant, Tree Root, Tsunami, King Snake, and Neural Network) which consistently appeared in all applications. On the other hand, other three patterns (Black Sheep, God Concern, and Copy Cat) appeared in small numbers, although uniformly throughout the case studies.

We also applied our concern-based detection technique to the corresponding AspectJ versions of the systems. Matching our intuition, the aspect-oriented (AO) solutions revealed fewer crosscutting patterns since crosscutting concerns in Java implementations had been better separated in AspectJ programs. The overall reduction of crosscutting patterns was around 43% (468 in Java vs. 266 in AspectJ). Particularly, AspectJ removed almost 90% of the Black Sheep pattern's instances; just two instances remained in all AO systems (against 18 in Java). On the other hand, only 6% of Tree Root instances were removed by the use of aspects (45 instances in AspectJ vs. 48 instances in Java). This result suggests that some crosscutting patterns, such as Tree Root, may not be easily addressed by the AO mechanisms of AspectJ.

4.3. Crosscutting patterns vs. design stability

This section assesses the correlation of specific crosscutting patterns and design stability. This part of the evaluation comprised three steps. First, we identified the most unstable classes by analysing real changes throughout the MM and HW software evolutions. Second, we detect each of the 13 crosscutting patterns by analysing all concerns in the 4th MM release (R4) and in the 9th HW release (R9). Considering that those applications evolved towards a more stable structure, R4 of MM was chosen as a representative of an intermediary one. In other words, it is neither the most unstable release (R1) nor the most stable one (R7). On the other hand, we selected R9 of HW as a representative (more stable) release in later stage of the software evolution. Finally, we contrasted the number of crosscutting patterns' instances found in each class with the list of unstable classes (first step).

Tables 5 presents 5 of the most unstable classes (top classes) and 5 of the most stable classes (bottom classes) for MobileMedia. This table also shows the number of changes (last column) and the number of crosscutting patterns (intermediary columns) per class of this system. Table 6 presents the same information for Health Watcher. It is easy to recognise that classes with higher occurrences of changes are also the locus of more instances of crosscutting patterns. For example, classes of MobileMedia (Table 5) changing in 5 out of 7 releases are affected by at least 6 instances of crosscutting patterns. On the other hand, no more than 5 instances of crosscutting patterns were found in the most stable classes of both systems.

Table 5. Stability vs. crosscutting patterns (MM)

Components MobileMedia R4	Crosscutting Patterns							# of Changes
	BS	Oct	CP	TR	Tsu	KS	CC	
BaseController	0	2	1	1	1	1	0	5
MediaAccessor	0	5	0	1	2	3	2	5
MediaController	2	6	1	6	2	5	2	5
MediaUtil	2	4	0	2	1	2	0	5
PhotoViewScreen	0	5	0	1	1	0	1	4
<<complete list is available at [3]>>								
AlbumListScreen	0	1	0	0	1	0	0	1
BaseMessaging	0	0	0	0	0	0	0	1
ControllerInterface	0	1	1	1	1	1	0	1
BaseThread	0	0	0	0	0	0	0	0
SplashScreen	0	0	0	0	0	0	0	0

Table 6. Stability vs. crosscutting patterns (HW)

Components Health Watcher R9	Crosscutting Patterns						# of Changes
	Oct	GC	HD	KS	NN	CC	
HealthWatcherFacade	4	1	0	1	2	2	6
HWServlet	1	2	0	0	1	1	6
UpdateHealthUnitSearch	3	2	0	0	1	1	6
UpdateComplaintData	2	1	0	0	1	1	5
GetDataForSearch	2	2	0	0	2	2	4
<<complete list is available at [3]>>							
Date	1	1	0	0	1	1	0
Observer	4	0	0	0	1	0	0
RMIFacadeFactory	1	0	1	0	0	0	0
Schedule	1	1	0	0	1	1	0
Situation	1	1	0	0	0	1	0

In order to investigate how each crosscutting pattern relates to design stability, we analysed the number of changes per component. Although our preliminary results are not statistically significant, we identified in our analysis possible correlations between some crosscutting patterns and design stability. For instance, interestingly, all four communicative concerns (Section 3.3) have shown the highest correlation with module changes. This finding might better explain why modules referencing common blocks of code were found to be main sources of instabilities in a recent case study [13].

Figure 6. Correlation of Tree Root and Climbing Plant to component changes

For illustration, Figure 6 (left) shows a scatter plot that correlates the number of Tree Root instances and the number of changes for each component in Health Watcher. The dashed line represents the ideal linear match. In other words, it plots a function $f(x) = c.x$ where x is the number of changes, $f(x)$ is the number of pattern's instances, and c is a constant defined by the average number of patterns divide by the average number of changes. The closer plotted points are to the dashed line, the greater the correlation between the target crosscutting pattern and component stability.

The correlation is not so apparent between design stability and crosscutting patterns of the other groups. In fact, we could barely find a weak correlation of those for inheritance-wise concerns (Section 3.2) in our study. For example, the scatter plot presented in Figure 6 (right) shows the results of Climbing Plant in MobileMedia as a representative of this group. The plotted points of this chart do not follow the dashed line at all. For crosscutting patterns in the other two groups (Sections 3.1 and 3.4), the correlation is somehow moderated. In fact, it is always placed between those two extremes, i.e., between the high correlation presented by communicative concerns and barely any correlation by inheritance-wise concerns.

5. Related work

Our work was inspired by the concern patterns introduced by Ducasse and colleagues [4]. They proposed a generic technique, called Distribution Map, to visualise and analyse the concerns of a system. Based on this technique, they identified four patterns of concerns: well-encapsulated, crosscutting, black sheep, and octopus. Our work complements their work by formalising octopus and black sheep and by refining crosscutting into eleven new patterns. We have also evaluated the proposed crosscutting patterns against design stability.

Several studies [11, 17, 18, 23] have proposed refactorings for modularizing specific types of crosscutting concerns. For example, roles [11] and concern types [17, 18] could be seen as preliminary steps towards the definition of crosscutting patterns. However, their descriptions (i) are not abstract and generic as required by universal catalogues of patterns and (ii) cannot be applied to either design artefacts or beyond the context of specific programming mechanisms. For example, names for roles [11] are based on a very restrict set of specific low-level concerns (e.g., roles defined by design patterns). In addition, concern types [17, 18] are usually tied to constructs of specific AOP languages (e.g., *declare throws clause* of AspectJ).

6. Concluding remarks

This paper formalised 13 recurring patterns of crosscutting concerns identified in three heterogeneous case studies (a Web-based system, a software product line, and a design pattern library) from different domains. The list of crosscutting patterns complements and refines previous considerations [1, 4, 11, 18, 23] regarding the problems of dealing with crosscutting concerns. It also presented an exploratory evaluation that investigated the most common crosscutting patterns in aspect-oriented and object-oriented systems. Our results indicated crosscutting patterns that are not easily addressed by typical aspect-oriented mechanisms as available in AspectJ (Section 4.2). Our evaluation also correlated specific crosscutting patterns to design stability (Section 4.3). It was found out that some crosscutting patterns, such as Tree Root, are strong indicators of change-prone classes. However, this correlation is not so evident in other patterns of crosscutting concerns like Climbing Plant.

Future work following our steps includes (i) the identification and formalisation of additional patterns of crosscutting concerns, (ii) analysis of typical combinations of the crosscutting patterns and the positive and negative impact of these combinations in the system design, (iii) tool support for detecting all documented crosscutting patterns, (iv) further empirical experimentation to confirm or refute our initial findings, and (v) analysis of whether and which crosscutting patterns are correlated to other software quality attributes, such as error-proneness.

7. References

[1] M. Bruntink, A. van Deursen, R. van Engelen, T. Tourwé. "An Evaluation of Clone Detection Techniques for Identifying Crosscutting Concerns". Proc. of the Int'l. Conference on Software Maintenance (ICSM), 2004.

[2] S. Chidamber and C. Kemerer. "A Metrics Suite for Object Oriented Design". IEEE Transactions on Software Engineering, 1994.

[3] Patterns of Crosscutting Concerns: Evaluation, Jan/09. http://www.lancs.ac.uk/postgrad/figueire/concern/patterns/

[4] S. Ducasse, T. Girba, and A. Kuhn. "Distribution Map". Proc. of the Int'l. Conference on Software Maintenance (ICSM), pp. 203-212, 2006.

[5] M. Eaddy *et al*. "Do Crosscutting Concerns Cause Defects?". IEEE Transactions on Software Engineering, 34(4), pp. 497-515, 2008

[6] E. Figueiredo *et al*. "On the Maintainability of Aspect-Oriented Software: A Concern-Oriented Measurement Framework". Proc. of the 12th European Conference on Software Maintenance and Reengineering (CSMR), 2008.

[7] E. Figueiredo *et al*. "Evolving Software Product Lines with Aspects: An Empirical Study on Design Stability". Proc. of Int'l Conf. on Software Engineering (ICSE), 2008.

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

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. "Design Patterns: Elements of Reusable Object-Oriented Software". Addison-Wesley, 1995.

[10] P. Greenwood *et al*. "On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study". Proc. of the European Conference on Object-Oriented Programming (ECOOP). Berlin, Germany, 2007.

[11] J. Hannemann, G. Murphy, and G. Kiczales. "Role-Based Refactoring of Crosscutting Concerns". Proc. of the 4th Int'l. Conference on AOSD, pp. 135-146, 2005.

[12] J. Hannemann and G. Kiczales. "Design Pattern Implementation in Java and AspectJ". Proc. of the 17th OOPSLA Conference, pp. 161-173. Seattle, 2002.

[13] D. Kelly. "A Study of Design Characteristics in Evolving Software Using Stability as a Criterion". IEEE Trans. on Software Engineering, 32(5), pp. 315-329, 2006.

[14] G. Kiczales *et al*. "Aspect-Oriented Programming". Proc. of the ECOOP Conference, pp. 220-242, 1997.

[15] Dexter C. Kozen. "The Design and Analysis of Algorithms". 1st edition, Springer, 1991.

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

[17] M. Marin, L. Moonen, and A. van Deursen. "An Approach to Aspect Refactoring Based on Crosscutting Concern Types". ACM SIGSOFT Software Engineering Notes, 30(4), pp. 1-5, 2005.

[18] Marin, M *et al*. "A Classification of Crosscutting Concerns". Proc. of the 21st IEEE international Conference on Software Maintenance (ICSM), 2005, pp. 673-676.

[19] R. Marinescu. "Detection Strategies: Metrics-Based Rules for Detecting Design Flaws". Proc. of the Int'l. Conf. on Soft. Maintenance (ICSM), pp. 350-359, Chicago, 2004.

[20] MobileMedia Project at SourceForge.net. Accessed 19/01/2009: http://sourceforge.net/projects/mobilemedia/

[21] M. Robillard and G. Murphy. "Representing Concerns in Source Code". Trans. on Software Engineering and Methodology (TOSEM), 2007.

[22] C. Sant'anna *et al*. "On the Modularity of Software Architectures: A Concern-Driven Measurement Framework". Proc. of the ECSA Conference. Spain, 2007.

[23] B. Silva, E. Figueiredo, A. Garcia, and D. Nunes. "Refactoring of Crosscutting Concerns with Metaphor-based Heuristics". Proc. of the CSMR workshop on Software Quality and Maintainability (SQM), 2008.

[24] P. Tarr, H. Ossher, W. Harrison, and S. Sutton. "N Degrees of Separation: Multidimensional Separation of Concerns". Proc. of the Int'l Conference on Software Engineering (ICSE), 1999.