# On the Support and Application of Macro-Refactorings for Crosscutting Concerns

**Bruno C. da Silva[1], Eduardo Figueiredo[2], Alessandro Garcia[3] and Daltro J. Nunes[1]**

[1] Informatics Institute – Federal University of Rio Grande do Sul (UFRGS) – Brazil

[2] Computing Department – Lancaster University, United Kingdom.

[3] Opus Research Group - LES, Informatics Department – PUC-Rio, Brazil.

{bcsilva,daltro}@inf.ufrgs.br, e.figueiredo@lancaster.ac.uk, afgarcia@inf.puc-rio.br

***Abstract****. Crosscutting concerns hinder software stability and reuse and, hence, refactorings have been proposed to modularise them using aspect-oriented programming technology. However, refactoring of crosscutting concerns is challenging and time-consuming because it involves many inter-dependent micro-refactorings. It may also be a repetitive task as recent studies have pointed out that most crosscutting concerns share a limited number of recurring shape patterns. This paper presents a family of macro-refactorings for modularising crosscutting concerns which share similar forms and patterns. It also proposes a complementary set of change impact algorithms which support designers on the decision whether to apply concern refactoring. We evaluate our technique by measuring the impact of refactoring 22 crosscutting concerns in two applications from different domains.*

## 1. Introduction

A concern is any critical or important consideration to one or more stakeholders involved in the software development and maintenance [Robillard 2007]. The reuse and stability of software modules is largely dependent on the ability of developers to wisely refactor the so-called *crosscutting concerns* [Eaddy 2008] into software modules, such as aspects [Kiczales 1997]. However, refactoring [Fowler 1999] of such concerns is a non-trivial, time-consuming software maintenance task for many reasons. First, crosscutting concerns entail many inter-related pieces of source code scattered through multiple modules. Second, these pieces might share some properties, which means that isolate, unordered use of existing module-driven micro-refactorings [Fowler 1999; Hanenberg 2003; Monteiro 2006] does not suffice and is counter-productive. Third, according to [Murphy-Hill 2009], about 40% of tool-initiated refactorings occurs in batches. They have observed that most of refactoring sequences are applied manually and are error-prone [Murphy-Hill 2009].

The situation is exacerbated in crosscutting concern modularisation as developers have to perform several co-dependent micro-changes [Silva 2009a]. Several fine-grained refactorings [Fowler 1999; Hanenberg 2003; Monteiro 2006] need to be applied sequentially in a short period of time to achieve the full concern modularisation. It was also recently found that concerns exhibit recurring categories of crosscutting shapes or patterns [Figueiredo 2009], thereby making repetitive the refactoring steps

associated with each crosscutting category. Given the widely-scoped nature of crosscutting concerns, the decision to refactor them (or not) with aspect-oriented programming (AOP) [Kiczales 1997] is not easy either. One of the key factors to consider is the degree of change impact [Greenwood 2007]. In addition, recent empirical studies [Figueiredo 2008; Greenwood 2007; Figueiredo 2009] have pointed out that refactoring crosscutting concerns with aspects is not always beneficial.

Several techniques have been proposed in the literature to apply aspect-oriented (AO) refactoring into object-oriented (OO) software [Iwamoto 2003; Hanenberg 2003; Hannemann 2005; Marin 2005; Monteiro 2006; Binkley 2006]. The problem is that they are not concern-wise and often require a huge list of disconnected transformations to modularise even a typical, simple crosscutting concern, such as the Observer design pattern [Gamma 1995]. They do not guide the designer to holistically implement the set of micro-changes related to the scattered elements constituting a crosscutting concern. Hence, it becomes difficult to choose and apply a set of fine-grained refactorings in a feasible order to achieve the concern modularisation.

In this context, this paper presents a family of concern-aware coarse-grained refactorings (or simply macro-refactorings) [Silva 2009a] based on AOP. They address the limitation of existing micro-refactorings (Section 2) by guiding the developers to modularise crosscutting concerns with correlated fine-grained code transformations. In particular, this paper provides two major contributions. It extends our previous work [Silva 2009a], which described two initial macro-refactorings for crosscutting concern modularisation. We provide a catalogue of macro-refactorings for thirteen recently-documented crosscutting patterns [Figueiredo 2009] (Section 3). These refactorings can be reused every time a concern matches the crosscutting pattern addressed by the refactoring. As part of the evaluation procedures, we also present a complementary set of change impact algorithms to support designers on the decision whether to apply concern refactoring or not (Section 4.1). Information about the change impact of refactoring candidates is required when reasoning about the feasibility and cost effectiveness of such task [Mens 2004]. We also provide a systematic evaluation of our concern-sensitive refactorings (Section 4) and concluding remarks (Section 5).

## 2. A Discussion of Aspect-Oriented Refactoring Techniques

Aspect-Oriented Programming (AOP) provides explicit mechanisms for improving the modularisation of otherwise crosscutting concerns through the notion of *aspects* [Kiczales 1997]. Refactoring is one of the essential techniques used to mitigate design flaws [Fowler 1999], such as crosscutting concerns. Aiming at improving quality attributes of software design, refactoring practices have emerged through the use of behaviour-preserving transformations over code units [Fowler 1999]. This section presents a brief review and discussion of available refactoring techniques which take into account the existence of AOP.

**Existing Categories of AO Refactorings.** AOP-related refactorings can be divided into three categories: first, OO refactorings that have been extended to become aspect-aware [Iwamoto 2003; Hanenberg 2003]; second, aspect-oriented (AO) refactorings particularly focused on AOP constructs [Garcia 2004; Monteiro 2006]; and, third, refactorings tailored for supporting extraction and modularisation of crosscutting concerns [Marin 2005; Hannemann 2005]. Our proposed refactorings fit in the third

category and are used to address crosscutting code as the one exemplified in Figure 1. Figure 1 shows code fragments realising the Exception Handling concern extracted from a Web-based system, called Health Watcher (HW) [Greenwood 2007]. Health Watcher is one of the target applications used in our study evaluation (Section 4). The two frames depicted in Figure 1 consist of exception handling blocks which are clones of crosscutting code spread through many modules of the HW system. The first one was encountered 28 times, while the second has 7 occurrences.

28x

```
try{
...
} catch (RemoteException e)
{
  throw new CommunicationException(e.getMessage());
}
```

7x

```
try{
...
} catch (RemoteException e)
{
  e.printStackTrace();
}
```

**Figure 1 – Fragments extracted from the EH concern of Health Watcher**

According to recent approaches for refactorings using AOP, there are a number of alternatives to restructure this code using aspects. In the case of exception handling blocks, we have some specific options: (i) Filho and his colleagues (2006) proposed a cookbook to aspectise exception handling code; and (ii) Binkley (2006) presented similar refactorings (*Extract Exception Handling*). However, these alternatives are specific to exception handling blocks and are tightly coupled to specific programming language mechanisms. Such refactorings are obviously specific to exception handling and cannot be applied to other forms of crosscutting concerns which are also implemented by replicated code. Alternatively, a designer could consider the reuse of fine-grained refactorings from existing catalogues, such as: (i) *Extract Code to Advice* and *Extract Pointcut Definition* [Garcia 2004]; (ii) *Extract Fragment into Advice* [Monteiro 2006]; (iii) Fowler's OO refactorings to prepare or to adapt the source code for application of AO refactorings; and (iv) refactorings to restructure the internals of aspects or to dealing with generalisation [Monteiro 2006].

**Lack of Concern-Aware Refactorings.** However, it is difficult to grasp from the variety of available fine-grained OO and AO refactorings the ones to compose a coarse-grained refactoring intended to modularise (or portion of) a concern. Moreover, in some cases we have to deal with imprecise definitions of refactorings and in many situations refactorings with overlapped intentions and similar names. In fact, fine-grained refactorings available in the literature [Iwamoto 2003; Hanenberg 2003; Binkley 2006] are usually defined without a standard and consistent terminology. Most of them [Garcia 2004; Monteiro 2006] address the same situation and have similar goals. Some examples include *Extract Exception Handling* [Binkley 2006], *Extract Pointcut* [Iwamoto 2003], *Extract Pointcut Definition* [Garcia 2004], and *Extract Advice* [Hanenberg 2003]. The selection of such a list of refactorings and their composition is not a trivial task and varies on different contexts. Furthermore, once designers have settled up a suitable composition of fine-grained refactorings to be carried out into a

specific context (for instance, the case of Figure 1), there is no guarantee they will remember of that specific composition and reuse it in another occasion.

## 3. Macro-Refactorings for Crosscutting Concern Patterns

This section presents a catalogue of macro-refactorings (Section 3.1) for modularisation of crosscutting concerns based on their recurring realisation patterns. These patterns were identified and documented by previous work [Figueiredo 2009].

### 3.1. Catalogue of Macro-Refactorings

A concern can manifest itself in a variety of different ways, defined by its *crosscutting pattern* [Figueiredo 2009]. Crosscutting patterns are symptoms to motivate the application of refactorings for crosscutting concerns because they represent harmful concern realisations. However, designers are not required to factor out every instance of a crosscutting pattern. For instance, there are crosscutting concerns strongly coupled to the base code that make hard or inappropriate their refactoring. In these cases, an attempt of refactoring would cause many widely-scoped changes. Some algorithms for change impact analysis are presented in Section 4 to support the refactoring decision, taking modification trade-offs into account.

**Table 1 – Crosscutting patterns and corresponding refactorings**

| Category | Refactoring Name | Recommended Action |
|---|---|---|
| Flat Crosscutting Shapes | Octopus | It aims at moving to aspects parts of classes composing the *body* or touched by *tentacles* of an Octopus concern. |
| | Black Sheep | It identifies classes implementing *slices* of the Black Sheep concern and modularises these slices into aspects. |
| | God Concern (*) | It tries to decompose the God Concern into several sub-concerns. Then, the modules members realising sub-concerns should be aspectised. |
| Inheritance-wise Concerns | Climbing Plant (*) | It eliminates the concern realisation from an inheritance tree. |
| | Hereditary Disease | Similar to Climbing Plant, but considering the existence of *disease-free nodes* (modules not realising the concern). |
| Communicative Concerns | Tsunami (*) | It minimises coupling among modules where one of them, called *wave source*, is coupled to all others, called *waves*. |
| | Tree Root | Inversely, it minimises coupling among modules where one of them, called *trunk*, receives incoming coupling connections from other modules, called *feeders*. |
| | King Snake | It aims at modularise a non-cyclic chain of coupling connections among modules realising a concern. |
| | Neural Network | It makes possible the aspectisation of modules composing a network of coupling connections among them. |
| Other Crosscutting Patterns | Copy Cat (*) | It removes replications of concern code in modules by the aspectisation of structural and behavioural *copies*. |
| | Dolly Sheep | |
| | Data Concern | It tries to better modularise concerns only composed of data (i.e., attributes and accessors operations). |
| | Behavioural Concern | Complementarily, it tries to better modularise concerns totally formed by behaviour (i.e., operations). |

*(*) Indicates representative refactorings to be detailed in the next sections.*

Table 1 shows macro-refactorings for crosscutting concerns in each pattern category. All the patterns are precisely described including examples in [Figueiredo 2009]. This table also summarises the recommended action for each refactoring. In the

case of Copy Cat and Dolly Sheep, just one macro-refactoring is presented since the latter is a specialisation of the former. Due to space constraints, we detail in the following sections a representative macro-refactoring (marked with * in Table 1) of each crosscutting pattern category. The presentation structure of the proposed macro-refactorings follows well-known refactoring catalogues [Fowler 1999; Monteiro 2006]. Basically, each macro-refactorings is presented in terms of a typical situation, a motivation, an abstract representation, recommended actions, and mechanics. Since a macro-refactoring is composed of micro-refactorings, we selected the following set of fine-grained refactorings from Fowler's and Monteiro's catalogues [Fowler 1999; Monteiro 2006] as our set of micro-refactorings: *Move Field/Method from Class to Intertype, Extract Fragment into Advice, Change Implements/Extends with Declare Parents, Move Method, Pull up Method/Field*. Therefore, we have available this set of refactorings to compose our macro-refactorings.

### 3.2. God Concern Refactoring

***Typical Situation*:** This symptom manifests when, in addition to being scattered and tangled over many modules, the concern also concentrates multiple intentions and functionalities. For instance, Figure 2 presents an abstract representation of God Concern and its respective refactoring. The left-hand side of this figure shows a God Concern instance. The shadow grey areas of this figure indicate parts of modules (represented by boxes) realising the concern under consideration. This figure highlights that God Concern is a widely-scoped crosscutting concern and requires a lot of functionality in its realisation.

***Motivation*:** God Concern indicates a design modularity flaw because (i) it represents a scattered and tangled concern and (ii) the concern concentrates multiple responsibilities.

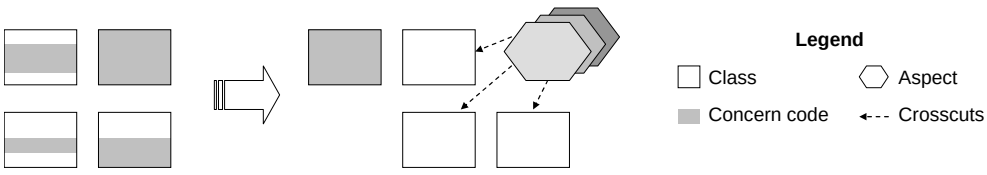


**Figure 2 – Abstract representation of the God Concern Refactoring**

***Recommended Action*:** The key action to address this problem is to try to decompose the God Concern into several sub-concerns; each of them representing a different modular slice of the God Concern. This decomposition aims at facilitating the modularisation of each portion of the God Concern in a separate aspect. The abstract representation of this refactoring shows how to modularise a God Concern instance using classes and aspects (right-hand side of Figure 2). The aspectisation should be done by using introductions, pointcuts, and advices as described below.

***Mechanics:***

1) **Identify possible concern decomposition:** check if it is possible to decompose the God Concern into new sub-concerns with well-defined intentions.
2) **Identify modules realising God Concern**.
3) **Transformation steps**

a. If new sub-concerns were decomposed from God Concern, then create at least one aspect for each new sub-concern.
b. If a class is totally dedicated to the concern, then optionally move it to an aspect. This step is not required because that class can be considered well-modularised.
c. If a class also participates in other concerns, then separate the God Concern parts and move them into an aspect.
d. For every attribute realising God Concern: *Move Field from Class to Intertype*.
e. For every method realising God Concern: *Move Method from Class to Intertype*.
f. For every code fragment realising God Concern: *Extract Fragment into Advice*.
g. For class extensions or interface implementations related to the concern realisation, apply *Change Implements/Extends with Declare Parents*.

### 3.3. Climbing Plant Refactoring

***Typical Situation:*** This symptom occurs when modules realising the concern are participating in an inheritance tree. The concern affects the root of an inheritance tree and propagates its structure to all children (also called *branches*) of this tree.

***Motivation:*** This crosscutting pattern introduces implicit dependency between modules via inheritance relationships. That is, changes in a branch can ripple through ancestral modules to other branches of the inheritance tree. For example, changing an overridden method could trigger changes in the abstract method definition and, as a result, further modifications might be required to other modules overriding the same method. Such ripple effect could be avoided if the concern is localised in one module (e.g., aspect).

***Recommended Action:*** The following actions are used to eliminate the concern realisation from an inheritance tree. Figure 3 presents the abstract representation before (left-hand side) and after (right-hand side) the application of the Climbing Plant refactoring steps. If the use of inheritance exists only for the concern realisation, then it should be moved to aspects and introduced back by intertype declaration (Alternative 1). Otherwise, inheritance is left in an object-oriented style (Alternative 2).

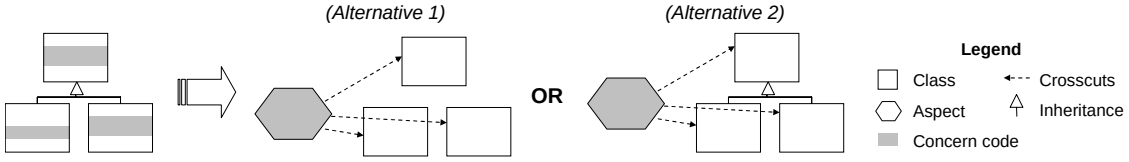


**Figure 3 – Abstract representation of the Climbing Plant Refactoring**

***Mechanics:***

**1) Identify the participating modules:** the *root* and *branches* of Climbing Plant.

**2) Transformation steps for the root**

a. If the root is completely assigned to the concern realisation, then optionally move it to an aspect.
b. If the root is assigned to more than one concern then it is required to separate the Climbing Plant concern and move it to an aspect. Attributes, methods, and code fragments realising the Climbing Plant root can be factor out to aspects using, respectively, *Move Field from Class to Intertype*, *Move Method from Class to Intertype*, and *Extract Fragment into Advice*.

**3) Transformation steps for branches**

a. Using a similar strategy of Step 2.b (above), each attribute, method, and code fragment realising a Climbing Plant branch has to be refactored. For instance, *Move Field from Class to Intertype* can be used to modularise concern-related attributes.
b. For class extensions and interface implementations related to the concern realisation, we can apply *Change Implements/Extends with Declare Parents*. Note that, an inheritance relationship should not be moved if it is not assigned to the Climbing Plant concern, i.e., this relationship may be assigned to other concerns.

### 3.4. Copy Cat Refactoring

***Typical Situation:*** This symptom occurs when replicated code implements the same concern. In other words, the given concern is implemented by similar pieces of code in different modules. Such replicated parts may be structural (field and method declarations) or behavioural (code fragments inside methods). This situation is similar to the *duplicated code bad-smell* proposed by Fowler (1999). However, Copy Cat refers to duplications related to a specific concern realisation [Figueiredo 2009].

***Motivation:*** These duplications can occur, for example, as a result of copy and paste practices and they increase the overall costs of maintenance activities [Fowler 1999]. 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. Such situation may affect the maintainability of the underlying concern. Examples like this should be eliminated by concentrating a single copy into an aspect and introducing this copy in several parts by means of aspectual mechanisms.

***Recommended Action:*** To eliminate the replications, structural copies could be localised into aspects and introduced back by inter-type declarations. For behavioural copies, it is necessary to use pointcuts to pick up the appropriate joinpoints and execute the copies' behaviours by means of advice. Figure 4 illustrates the Copy Cat Refactoring. Basically, the replicated concern code (labelled 'a' in the left-hand side of Figure 4) is moved to an aspect (right-hand side) which, in turn, introduces the code back to the appropriate classes.

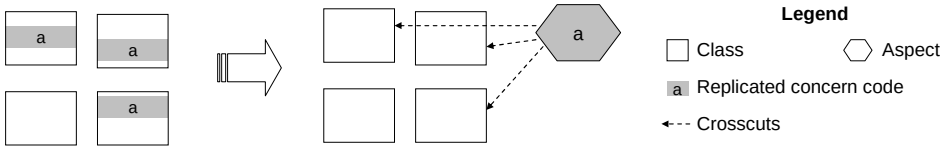


**Figure 4 – Abstract representation of the Copy Cat Refactoring**

***Mechanics:***

**1) Identify the participating modules:** The key elements to be identified are modules implementing replicated concern code. We also need to distinguish structural copies (field and method declarations) from behavioural ones (code fragments inside methods).

**2) Refactoring steps for structural copies:**

**a.** If modules have replicated methods realising the same concern and are involved in inheritance relationships, then *Pull up Method* can be applied to move one copy of the replicated concern method to a superclass. The target superclass should realise the same concern. The replicated methods should be available to all

subclasses relying on them. The remaining copies of these concern-related methods can then be removed.

**b.** If modules involved in inheritance relationships have replicated attributes realising the same concern, then *Pull up Field* can be used into one of the copies to have the replicated concern attribute available in just one place. Attributes realising the concern have to be accessible to subclasses which rely on them. Finally, the remaining copies can then be removed.

**c.** If modules which have concern copies are not involved in an inheritance tree, then *Move Field from Class to Intertype* can be applied for each replicated concern attribute and *Move Method from Class to Intertype* for each replicated method.

**3) Refactoring steps for behavioural copies:**

**a.** If the modules which have replicated code fragments are involved in inheritance relationships, then *Extract Method* can be used to create a new method with the replicated code fragment. Then, move the newly created concern method to a superclass following Step 2.a above.

**b.** *Extract Fragment into Advice* can be used to extract pointcuts which pick up the joinpoints for behavioural copies. Move copies to aspect by creating an advice to execute the corresponding behaviour. Alternatively, *Extract Method* is used to expose replicated code fragments and then, *Move Method from Class to Intertype*.

### 3.5. Tsunami Refactoring

***Typical Situation:*** This symptom occurs when modules are coupled to each other due to the concern realisation. Moreover, there is a core module, name *wave source*, which direct or indirect connects to all other participating modules (called *waves*). This situation resembles wave propagation of coupling connections (left side of Figure 5).

***Motivation:*** The high coupling level caused by this crosscutting pattern is a typical modularity flaw. The *wave source* is a highly coupled module which is difficult to comprehend and maintain since it depends on several pattern's participating modules.

***Recommended Action:*** After identifying the *wave source* and composing *waves*, we have to modularise the scattered concern code (Figure 5). Refactoring a Tsunami-forming module is optional if this module is fully dedicated to the concern realisation.

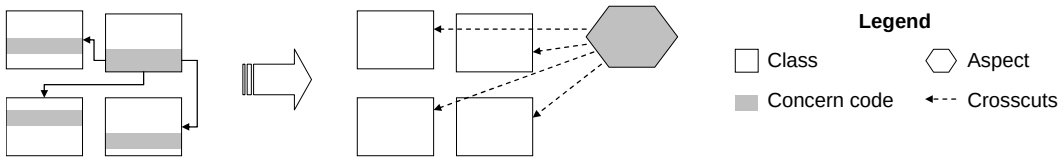


**Figure 5 – Abstract representation of the Tsunami Refactoring**

***Mechanics:***

**1) Identify the participating modules:** the *wave source* and *waves*.

**2) Transformations steps to be applied to both wave source and waves:**

a. If a module is completely assigned to the concern realisation, then optionally move it to an aspect.

b. For every attribute realising Tsunami: *Move Field from Class to Intertype.*

c. For every method realising Tsunami: *Move Method from Class to Intertype*.

d. For every code fragment realising Tsunami: *Extract Fragment into Advice*.

## 4. Evaluation

This section describes a systematic evaluation of the proposed macro-refactorings (Section 3). We applied our technique to two medium-sized applications, named Health Watcher [Greenwood 2007] and Mobile Media [Figueiredo 2008]. Health Watcher (HW) is a Web-based information system for supporting healthcare-related complaints. Mobile Media (MM) is a software product line for handling data on mobile devices. We selected 22 concerns from the two target systems and classified them according to 13 crosscutting patterns [Figueiredo 2009]. One concern can be classified in more than one pattern instance. From this analysis, 42 instances of crosscutting patterns were found.

### 4.1. Change Impact Analysis for Concern-Aware Refactorings

As part of the evaluation procedures, we have elaborated algorithms to compute measures variations over refactoring candidates. This strategy was followed to support change impact analysis and also to enable comparisons after the refactorings application. A programmer could also alternatively use such algorithms to decide for the refactoring or not. The impact is computed based on three concern metrics proposed by [Sant'anna 2003]: Concern Diffusion over Components (CDC) which counts the number of primary modules whose main purpose is to contribute to the implementation of a concern; Concern Diffusion over Operations (CDO) which counts the number of operations whose main purpose is to contribute to the implementation of a concern; and, Concern Diffusion over Lines of Code (CDLOC ) which counts the number of transition points for each concern through the lines of code.

We have defined an algorithm for each macro-refactoring of Table 1 (Section 3). Besides, these algorithms use sub-routines to compute the impact of fine-grained refactorings. Note that refactoring for crosscutting pattern modularisation is composed of fine-grained refactorings. The algorithms for impact analysis require about seven sub-routines to support the evaluation of the measurement variations. Due to space constraints, we show only two algorithms (Listing 1 and Listing 2) as representative ones. The design of other algorithms follows a similar rationale. All of them follow the transformation steps defined for the corresponding macro-refactoring (Section 3). For instance, if a condition appears on the transformation steps, a similar conditional structure is defined by the algorithms. Moreover, calls to sub-routines correspond to points where a transformation step requires the use of micro-refactoring.

Listing 1 shows a routine which evalutes the impact of the micro-refactoring *Extract Fragment into Advice*. This routine computes the CDO and CDLOC values according to the concrete instance of the crosscutting pattern under consideration. The CDO value is decreased by 1 unit if the operation which contains the fragment does not have another fragment related to the concern (first *IF* in Listing 1). On the other hand, CDO is increased by 1 unit if a new advice is created just for the extracted fragment (second *IF*). Regarding CDLOC, this measure is decreased by 2 units depending on the two situations indicated by the last *IF* and *FOR EACH* declarations, respectively.

The algorithm depicted in Listing 2 evaluates the impact of the God Concern Refactoring presented in Section 3.2. The CDC value varies at the beginning and at the end of this algorithm. It is increased by 1 unit for each new concern decomposed from God Concern; each new concern corresponds to a new aspect which modularises it. This

value is also decreased by 1 unit for each refactored class which has all its concern-related parts moved to aspects. Additionally, this algorithm calls subroutines concerned with micro-refactoring steps, such as *Extract Fragment into Advice* (Listing 1).

```
PROCEDURE evaluateImpactExtractFragmentIntoAdvice(var CDO, var CDLOC)
BEGIN
   IF the operation which contains the fragment does not have another fragment related
to the concern THEN
      CDO <- CDO - 1;
   END-IF
   IF a new advice will be created just for the extracted fragment THEN
      CDO <- CDO + 1;
   END-IF
   IF there is no code related to the concern immediately adjacent to the extracted
fragment THEN
      CDLOC <- CDLOC - 2;
      FOR EACH local variable used only by the fragment AND with no adjacent code
related to the concern DO
         CDLOC <- CDLOC - 2;
      END-FOR
   END-IF
END
```

**Listing 1 – Impact routine for Extract Fragment into Advice**

```
PROGRAM ImpactAnalysis_GodConcernRefactoring
var
   n_modified_op, n_fragments, n_methods: Integer;
   CDC, CDO: Integer
   CDLOC: Integer;
BEGIN
   CDC, CDO, CDLOC, n_modified_op, n_fragments, n_methods <- 0;
   IF the God Concern will be decomposed into new ones THEN
      FOR EACH new concern DO
         CDC <- CDC + 1;
      END-FOR
   END-IF
   FOR EACH module to be refactored DO
      IF there is inheritance or interface implementation related to the concern THEN
         evaluateImpactMoveImplementsExtends(CDLOC);
      END-IF
      FOR EACH attribute related to the concern DO
         evaluateImpactMoveFieldIntoAspect(n_modified_op, n_fragments, n_methods, CDO,
CDLOC);
      END-FOR
      FOR EACH method related to the concern DO
         evaluateImpactMoveMethodIntoAspect(CDLOC);
      END-FOR
      IF there is code fragment related to the concern THEN
         FOR EACH fragment DO
            evaluateImpactExtractFragmentIntoAdvice(CDO);
         END-FOR
      END-IF
      IF every concern-related member or code fragment was eliminated from the module
THEN
         CDC < - CDC - 1;
      END-IF
   END-FOR

END
```

**Listing 2 – Impact analysis for God Concern refactoring**

## 4.2. Evaluation of Macro-Refactorings for Crosscutting Concerns

Table 2 shows partial results of the impact analysis and refactoring focusing on five concerns (the ones with the most interesting results). The complete results are available at a supplementary website [Silva 2009b]. The numbers in Table 2 indicate variations of the 3 concern metrics in two situations: first, according to our algorithms for impact analysis (Section 4), before the application of macro-refactorings; and

second, according to the refactoring itself, collecting data before and after the corresponding application to modularise the target concern. Hence, the higher the negative numbers for CDC, CDO, and CDLOC, the better the concern modularity results. By analysing these data, we can verify that, in general, both the impact analysis and refactoring indicate improved separation of concerns. That is, in most cases (pair Concern/Crosscutting Pattern) the measurement varies negatively or stays neutral.

**Table 2 – Results from impact analysis and refactoring of HW and MM**

| System | Concern | Crosscutting Pattern | Measurement Variations | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | Impact Analysis | | | Refactoring | | |
| | | | CDC | CDO | CDLOC | CDC | CDO | CDLOC |
| HW | Abstract Factory | Climbing Plant | - | - | - | - | - | - |
| | Command | **Copy Cat** | **0** | **1** | **0** | **0** | **1** | **0** |
| | | Climbing Plant | - | - | - | - | - | - |
| | Observer | Octopus | -14 | -15 | -54 | -14 | -15 | -50 |
| | | Copy Cat | -4 | -11 | -44 | -4 | -11 | -44 |
| | | Climbing Plant | -5 | -9 | -42 | -5 | -9 | -42 |
| | | Hereditary Disease | -2 | 0 | -4 | -2 | 0 | -4 |
| | State | Climbing Plant | - | - | - | - | - | - |
| MM | Label Media | Tsunami | -6 | 4 | -108 | -4 | 5 | -102 |
| | | **Climbing Plant** | **-1** | **1** | **-14** | **0** | **1** | **-14** |

**Refactoring Tiny Crosscutting Patterns does not Pay off.** Table 2 shows three instances of the Climbing Plant crosscutting pattern where the impact analysis and refactoring were not carried out. These three Climbing Plant instances refer to the *Abstract Factory*, *Command*, and *State* concerns. These pattern instances were not factored out because they involve very few elements of a concern. Hence, the effort of refactoring these tiny pattern instances would represent overreaction with respect to insignificant gains in terms of separation of concerns. This observation is backed up by previous studies on aspect-oriented pattern implementations [Garcia 2005].

**Positive Variation of CDO**. In the HW study, an instance of Copy Cat (the *Command* concern in bold, Table 2) presented a positive variation in its CDO value. This situation is explained by the fact that some refactoring steps create either new methods to expose joinpoints or new advices to introduce extracted code fragments from methods. In the HW particular case, the latter (new advice) is responsible for the increase of CDO. This situation also appears in the MM study with the *Label Media* concern. In both cases there was one extra piece of advice and, therefore, CDO is increased by 1 unit.

**Conflicting Measurements of Impact Analysis and Refactoring.** Measurements for change impact analysis and refactoring do not match for two crosscutting patterns: Octopus and Tsunami, considering the *Observer* and *Label Media* concerns respectively. In those cases (shaded in Table 2) the algorithms for change impact analysis suggest better results compared to the actual application of refactorings. These differences highlight the existence of specific code fragments of a concern which could not be refactored in practice. We found two of these situations summarised as follows: (i) concern-related code fragments which do not match any refactoring step, and (ii) concern-related code fragments which designers decided to not refactor. For instance, in the latter case the designer realised that a transformation would either raise new design flaws or it could negatively impact on the modularisation of other modules or concerns.

It is expected that the impact analysis does not always give exactly the same numbers of the refactoring in practice. In fact, it is hard to define a precise algorithm given the widespread characteristic of some crosscutting concerns and its several refactoring possibilities. Nonetheless, we could not find many significant differences for the most cases. In other words, there was not big discrepancy taking into consideration the complexity of the problem at hand. It is important to note that even with small differences, the overall trend remained the same as followed discussed.

**Widely-Scoped Refactoring Favours Concern Modularity.** We also investigated the number of modules involved in the application of macro-refacorings for each crosscutting pattern. Then we analysed the measurement variation per number of refactored classes on each identified crosscutting pattern. Confirming expectations, we have observed that a macro-refactoring restructuring a pattern composed of more modules performs better than others restructuring fewer modules. Figure 6 supports this observation by showing charts for the *Observer* and *Label Media* concerns (from HW and MM systems, respectively). Each point in the charts represents a measurement for CDC, CDO, or CDLOC, varying in the *y* axis. The *x* axis expresses the number of refactored modules in a crosscutting pattern.

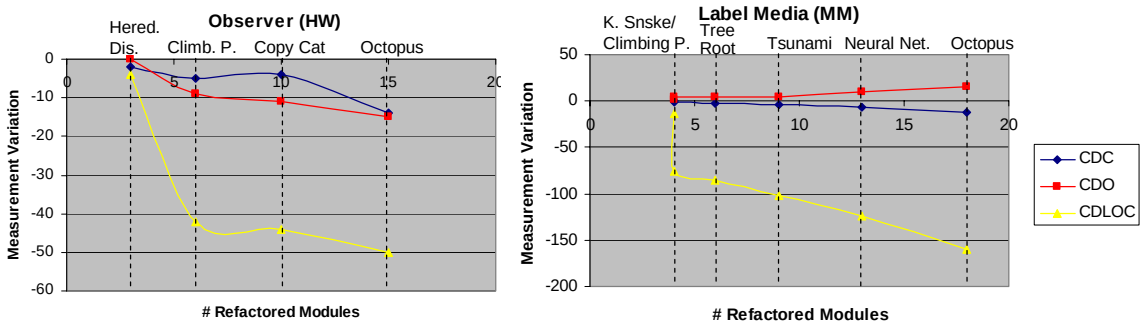


**Figure 6 – Measurement variation per refactored modules in a pattern**

Taking the *Observer* concern into account (left-hand chart), the CDC variation is -14 for the Octopus pattern (Table 2) when 15 classes are refactored. The values corresponding to CDO and CDLOC in the Octopus case also present a high variation (-14 and -50, respectively). By contrast, the refactoring of Hereditary Disease involved a smaller number of refactored classes (only 3). Moreover, we observe a small variation for all concern metrics (CDC, CDO and CDLOC) in the Hereditary Disease case. The variations of CDC, CDO and CDLOC are -2, 0, and -4, respectively. Therefore, we verify that, in this particular situation, the Octopus Refactoring performed better than the Hereditary Disease one since the former involved more modules than the latter.

The CDO variation of *LabelMedia* (second chart of Figure 6) could be seen as an exception to this rule. In fact, CDO presents a slightly increase when more modules require refactoring. This positive variation of CDO is due to new the creation of new advices as discussed earlier in this section. However, even with the positive variation of CDO, refactoring of more modules is still the best option when we consider trade offs of all three metrics. For instance, both charts of Figure 6 show that the CDC and CDLOC variations express better values for separation of concerns when more modules are involved in the macro-refactoring. This reflects the reduction of concern scattering and tangling when the number of refactored modules increases.

## 5. Conclusions and Future Work

This paper proposed macro-refactorings aiming to support the modularisation of concerns matching some pre-defined crosscutting patterns [Figueiredo 2009]. Our macro-refactorings extend our previous work [Silva 2009] and are composed of micro-refactorings [Iwamoto 2003; Hanenberg 2003; Hannemann 2005; Marin 2005; Monteiro 2006; Binkley 2006]. They provide a reusable set of transformations to be applied in recurring categories of crosscutting concerns. Besides, we have proposed algorithms to enable the evaluation on the change impact of refactorings which can also help designers on the decision of selecting and applying a particular refactoring. These algorithms rely on three metrics for separation of concerns in order to assess the change variation.

We evaluate the macro-refactorings by performing an exploratory study involving two target systems – Mobile Media and Health Watcher (Section 4). The results indicated that our macro-refactorings can be successfully applied in different concerns from those two systems. It was also possible to verify a better concern modularisation according to the three employed metrics. Particularly, we observed that our refactoring technique allows the composition and reuse of micro-refactorings in a simple way to modularise recurring categories of crosscutting concerns. Additionally, our algorithms for change impact have shown to be good indicators of refactoring activities allowing designers to reason about the trade-offs and cost effectiveness before actually applying refactorings. For instance, our comparison of the measurement variations for the refactoring application and for impact analysis algorithms indicates that they have similar results.

For future and ongoing work, we envision (i) the extension of our technique in order to support further crosscutting patterns which may be catalogued (ii) the automation of refactoring steps and change impact algorithms and (iii) new experimental studies to support or refute our preliminary findings. For example, we have learned that it is also important to consider in future studies some information not used in our current impact analysis such as the inter-dependence of modules through different concerns. Our previous experience in AO software assessment [Garcia 2005; Greenwood 2007; Figueiredo 2008] has indicated that it is an important work direction.

## References

Binkley, D. *et al*. (2006). Tool-Supported Refactoring of Existing Object-Oriented Code into Aspects. *IEEE Transactions on SW. Eng.*, Los Alamitos, USA, v.32, p.698–717.

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

Figueiredo, E. *et al*. (2008). “Evolving Software Product Lines with Aspects: An Empirical Study on Design Stability”. In: Proc. of the 30th ICSE, pp. 261-270. Leipzig, Germany, 10-18 May.

Figueiredo, E. *et al.*. (2009). “Crosscutting Patterns and Design Stability: an Exploratory Analysis”. In: Proc. of the 17th ICPC, Vancouver, Canada.

Filho, F. *et al*. (2006). “Exceptions and Aspects: the Devil is in the Details”. In: Proc. of the 14th FSE, New York, USA. ACM, p.152–162.

Fowler, M.; Beck, K.; Brant, J.; Opdyke, W. and Roberts, D. (1999). Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional.

Gamma, E. *et al*. (1995). Design Patterns: Elements of Reusable Object-Oriented Software. Boston, MA, USA: Addison-Wesley Longman Publishing.

Garcia, V. *et al*. (2004). "Manipulating crosscutting concerns". In: Latin American Conference on Patterns Languages of Programming (SugarLoafPlop'04).

Garcia, A. *et al*. (2005). "Modularizing Design Patterns with Aspects: A Quantitative Study". In: 4th AOSD, Chicago, USA, 14-18 March 2005.

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

Hanenberg, S.; Oberschulte, C. and Unland, R. (2003). "Refactoring of Aspect-Oriented Software". In: Net.ObjectDays Conference (NODE'03).

Hannemann, J.; Murphy, G.; Kiczales, G. (2005). "Role-based refactoring of crosscutting concerns". In: 4th AOSD, New York, USA. ACM, p.135–146.

Iwamoto, M. and Zhao, J. (2003). "Refactoring Aspect-Oriented Programs". In: Int'l. Workshop on Aspect-Oriented Modeling at UML'03.

Kiczales, G. *et al*. (1997). "Aspect-Oriented Programming". Proc. of the European Conference on Object-Oriented Programming (ECOOP), pp. 220-242.

Marin, M.; Moonen, L. and Deursen, A. (2005). An Approach to Aspect Refactoring Based on Crosscutting Concern Types. *Software Engineering Notes*, v.30, n.4, p.1–5.

Mens, T.; Tourwé, T. (2004). A Survey of Software Refactoring. *IEEE Transactions on Software Engineering*, Piscataway, NJ, USA, v.30, n.2, p.126–139.

Monteiro, M. and Fernandes, J. (2006). Towards a Catalogue of Refactorings and Code Smells for Aspectj. *Transactions on Aspect Oriented Software Development (TAOSD)*, Lecture Notes in Computer Science, n.3880, p.214–258.

Murphy-Hill, E.; Parnin, C. and Black, A. P. (2009). "How We Refactor, and How We Know It". In Proc. of the 14th Int'l Conf. on Software Engineering, New York, USA.

Robillard, M. and Murphy, G. (2007). Representing Concerns in Source Code. *Transactions on Software Engineering and Methodology* (TOSEM), v. 16.

Sant'anna, C.; Garcia, A.; Chavez, C.; Lucena, C. and Staa, A. (2003). "On the Reuse and Maintenance of Aspect-Oriented Software: an assessment framework". In: XVII Brazilian Symposium on Software Engineering.

Silva, B.; Figueiredo, E.; Garcia, A. and Nunes, D. (2009). Refactoring of Crosscutting Concerns with Metaphor-Based Heuristics. *Electronic Notes on Theoretical Computer Science*, v.233, p.105–125.

Silva, B.; Figueiredo, E.; Garcia, A. and Nunes, D. (2009). Macro-Refactorings: Evaluation. Jun/2009. http://www.inf.ufrgs.br/~bcsilva/macrorefactoring_evaluation