# A Classification of Crosscutting Concerns

**Marius Marin**
*Software Evolution Research Lab*
*Delft University of Technology*
*The Netherlands*
*A.M.Marin@ewi.tudelft.nl*

**Leon Moonen**
*Software Evolution Research Lab*
*Delft Univ. of Technology & CWI*
*The Netherlands*
*Leon.Moonen@computer.org*

**Arie van Deursen**
*Software Evolution Research Lab*
*CWI & Delft Univ. of Technology*
*The Netherlands*
*Arie.van.Deursen@cwi.nl*

**Abstract**

*Refactoring software to apply aspect oriented solutions requires a clear understanding of what are the potential crosscutting concerns and which aspect solutions to replace them with. This process can benefit from the recognition of recurring generic concerns and their reusable aspect solutions.*

*In this paper, we propose a classification of crosscutting concerns in* sorts *based on the analysis of various refactoring efforts. We discuss how* sorts *help concern understanding and refactoring, how they support the identification of crosscutting concerns, and how they can contribute to the evolution of aspect languages.*

## 1. Introduction

The identification and refactoring of crosscutting concerns in legacy code aims at improving the evolvability of existing systems. To achieve this goal, it is important to understand what are the crosscutting concerns in a system, and how their modularization can be improved through aspect refactoring.

Existing work in area of aspect refactoring proposed a significant number of examples of crosscutting functionality and associated aspect-oriented solutions, such as logging, design patterns [3, 2], aspectizable interfaces [9], and transaction management and business rules in enterprise applications [5]. However, these are generally small demonstrative examples that lack the structure and organization that would allow their use for recognition and understanding of specific crosscutting functionality. Furthermore, in many of the examples more than one crosscutting concern is involved and the same crosscutting concerns occur in various contexts. For instance: the distinctive crosscutting elements in the Observer pattern are the roles (Subject and Observer) superimposed to the classes participating in the implementation of the pattern, and the consistent behavior of notifying the observers required from the methods changing the state of the Subject object. The same role superimposition is present in other design patterns, like Visitor, which defines crosscutting roles (Element/Node) to accept visitors. Similarly, an authentication mechanism would rely on the consistent behavior of the methods requiring authentication to actually call a verification method.

Our investigations show that *role superimposition* and *consistent behavior* are a kind of generic concerns that come up repeatedly in many of the proposed refactorings. In spite of this, they are not emphasized in literature as recurring distinctive elements to which a generic reusable aspect solution can be associated. As a result, it is harder to identify them in new contexts and to apply a consistent aspect solution.

This paper aims to fill this gap by proposing a classification system for crosscutting concerns that distinguishes recurrent, atomic concerns as *sorts*. Sorts are described by a number of distinctive properties which will help in recognizing them and in guiding their refactoring towards an aspect-oriented solution. We believe that such an approach will improve aspect-based reasoning about the legacy code by making the developer aware of the possible crosscutting concerns and their generic characteristics and symptoms.

## 2. Sorts of Crosscuttingness

Crosscutting concern *sorts* [6] are generic descriptions of crosscutting functionality that can be classified based on three main characteristics:

- intent of the concern (behavioral, design or policy requirements);
- legacy (non-aspect) implementation idiom;
- (desired) aspect language mechanism that supports the modularization of the sort's concrete instances.

An important property of a concern sort is its *atomicity*: each sort is associated with the smallest unit that can be used to individually express and modularize a crosscutting concern. I.e. there is one (desired) aspect language mechanism to address the specific crosscutting functionality of a sort, and one associated legacy implementation idiom. Consequently, the implementation of a concrete crosscutting concern can be expressed as a combination of one or more sort instantiations. We add new sorts to our list whenever they (1) cannot be composed of elements already in the list, and (2) they cannot be split in smaller units. Note that not all sorts need to be associated with a concrete mechanism in an existing aspect language, they can also refer to *desired* mechanisms which can help language designers to evolve their languages.

We propose the following template for describing a sort. Although the description is language specific, using AspectJ

| | |
|---|---|
| **Name** | Consistent behavior |
| **Intent** | Implement consistent behavior for a number of elements that can be captured by a natural pointcut. The enforced consistent behavior is a precise step in the execution of each method in the set defining the formalized context. |
| **OO idiom** | Method calls invoking the desired functionality |
| **Aspect mechanism** | Pointcut to map the formalized context, such as specific methods in a class hierarchy, and Advice to encapsulate the desired functionality |
| **Instances** | Authentication; Wrap service level exceptions of business services into application level exceptions [7]; Notify listeners (Observer pattern); Log exception throwing events in a system[5]. |

**Figure 1. Consistent Behavior Sort**

and Java as reference languages, complementary descriptions could be made for other languages as well.

Our classification gives a **name** to each sort to develop a common language for referring to instances of crosscutting functionality. Next, we describe the sort's **intent** in terms of behavioral, design, and/or policy requirements. An **object-oriented idiom** describes the typical (Java) implementation of the sort's concerns. We associate the sort with a single **aspect mechanism** that can be used to refactor the sort's instances (and thus their idiomatic object-oriented implementation). For each sort, we discuss a number of **instances** to enhance the understanding of sorts through examples.

In our classification, the *aspect language mechanisms* provide the consistent criterion to classify the crosscutting concerns, and establish the level of granularity for this classification. An *aspect mechanism* is a minimal combination of constructs that can be used to aspectize a concern, as, for instance, *static introduction*, typically implemented by *declare parents* and *inter-type declaration* constructs.

A number of elements to additionally describe a sort are specific to the aspect mechanism addressing the sort, and, thus, shared by a number of sorts. These elements include **risks, limitations and benefits** of migrating the instances of a sort from a legacy implementation to an aspect oriented solution. In addition, the behavior preservation constraints require to analyze the **testing implications** that follow from each refactoring.

## 3. Example: Consistent Behavior

This section discusses in detail *Consistent Behavior*, one of the proposed sorts. Figures 1 and 2 summarize the elements characterizing this sort.

The purpose of concerns of the *consistent behavior* sort is to enforce and ensure that specific functionality is consistently executed by a number of methods. To exclude ordinary delegation of functionality, the set of elements which should behave consistently should be captured by a natural pointcut

| | |
|---|---|
| **Risks, limitations** | Advice constructs in a privileged aspect can break encapsulation; High degree of tangling might prevent refactoring; Anonymous classes cannot be referred consistently; Calls to the super's functionality cannot be migrated into advice; |
| **Benefits** | Improved modularity; Reduced code size; |
| **Testing implications** | Fault model and test adequacy for pointcut and advice constructs [1] |

**Figure 2. Pointcut and Advice specific issues**

definition, i.e., these elements should follow a regular pattern and not be a random selection of all elements.

The way of implementing such concerns in, for example, Java, is to call a method implementing the desired functionality from the scattered places where we want this consistent behavior. This idiom is exploited by the fan-in analysis technique for aspect mining, which specifically looks for such scattered calls [7].

The AspectJ pointcut and advice mechanism captures and modularizes such symptoms of enforced consistent behavior like scattered calls. The benefits of aspectizing concerns of this sort are in terms of concern localization and modularization, but also in reliability: elements in the targeted context will not be forgotten from implementing the desired behavior.

The practical activities we have conducted to refactor instances of this sort in real applications [1] revealed some risks and limitations of the aspect solutions. One of the risks is that the joinpoints captured by the defined pointcut include intentional omissions in the original application. These omissions should be checked. Other risks are that sophisticated pointcuts or preliminary refactorings might be required in some cases to unplug the crosscutting functionality before it can be refactored into an aspect solution.

Limitations include the lack of direct support for capturing anonymous classes in a pointcut, or to refer the super's functionality of the advised method from within the advice. Our previous work [1] contains detailed discussions of these risks and limitations for a number of larger refactorings.

The testing implications related to the employed aspect language feature comprise the faults for the pointcut and advice constructs [1]. These include the use of a wrong primitive pointcut, a wrong type pattern in the pointcut, or logical errors in the pointcuts' conditions. Possible faults for the advice could be in its specification or precedence, as well as its code that can break the advised method, or prevent it to comply with postconditions or class invariants.

## 4. Thirteen Canonical Sorts

We propose an initial set of canonical sorts based on the previous considerations. The sorts and their attributes are shown in Table 1 and are grouped by the aspect mechanism that addresses the specific crosscutting functionality of the sort.

| Sort | Intent | Object-oriented Idiom | Aspect mechanism | Instances |
|---|---|---|---|---|
| Consistent Behavior | Implement consistent behavior as a controlled step in the execution of a number of methods that can be captured by a natural pointcut. | Method calls to the desired functionality | Pointcut and advice | Log exception throwing events in a system; Wrap/Translate business service exceptions [7]; Notify listeners; Authorization; |
| Contract enforcement | Comply to design by contract rules, e.g., pre- and post-conditions checking | Method calls to method implementing the condition checking | Pointcut and advice | Contract enforcements specific to design by contract |
| Entangled roles | Extend a method with a secondary role or responsibility which is entangled with its primary concern | Implement a method with (entangled) functionality that belongs to a different concern than main concern of that method | Pointcut and around advice that overrides the method | Roles overlapping in Swing, such as view and controller; make methods belonging to the view role implement controller logic. |
| Redirection Layer | Define an interfacing layer to an object (add functionality or change the context) and forward the calls to the object | Declare a routing layer (decorator/adapter), and have methods in this layer to forward the calls | Pointcut and around advice | Decorator pattern, Adapter pattern [3]; Local calls redirection to remote instances (RMI) [8]; |
| Add Variability | Use method objects to pass a method as a parameter | Build and pass objects of (anonymous) types implementing a single, specific action: *ActionListener.actionPerformed()*, *Command.execute()*, *Runnable.run()* | Pointcut and around advice that creates the method object | Concurrent access - Thread safety, Authorization[5]; Callbacks in GUIs (e.g., ActionListener(s)) |
| Expose Context | Expose the caller's context to a callee by passing information to each method in the call stack to that callee (aka Wormhole [5]). | Add arguments to each method in the call stack | Pointcut and advice, where the pointcut collects the context to be passed | Transaction management, authorization [5]. |
| Role superimposition | Implement a specific secondary role or responsibility | Interface implementation, or direct implementation of methods that could be abstracted into an interface definition | Introduction mechanisms | Roles specific to design patterns: Observer, Command, Visitor, etc.; Persistence [7] |
| Support classes for role superimposition | Make the relationship between classes explicit (through nested classes) to superimpose a role (to a hierarchy) | Nested classes implementing a role/responsibility | Not supported; Desired: introduction for nested classes | Undo concern [1] |
| Policy enforcement | Impose a policy to a group of elements in the system | Not supported by language but by documentation/comments | Declare warning / error mechanisms | Limit access to a given functionality, e.g., accessing AWT functionality from EJB components [5] |
| Exception propagation | Propagate an exception for which neither the method nor its callers have an appropriate response. | Exception is propagated to callers | Use declare soft mechanism (risks: identity of exception is lost) | Checked SQLException thrown from methods in the JDBC API . |
| Declare *throws* clause | Add a specific exception to the *throws* clause of all the methods within a formalized context | Directly add the new *throws* clause | Not supported; Desired: declare throws mechanism [8] | Add RMI specific exception [8]; Add transaction exceptions [4]. |
| Design enforcement | Enforce design, such as classes in an hierarchy must declare no-args constructors | Not supported by language but by documentation/comments | Not supported | Persistence [7]; Bean objects |
| Dynamic behavior enforcement | Enforce rules for object use, like before-use initialization and post-use clean-up | Not supported by language but by documentation/comments | Not supported | Lifecycle [7]; Replace finalizers with methods to be invoked. |

**Table 1. Sorts of crosscuttingness.**

The *Contract enforcement* and *Consistent behavior* sorts share a number of characteristics; however, they refer to concerns with different *intents*: contract enforcement ensures conditions due to relations between a method and its callers.

Concerns of the *Entangled roles* sort are specific to roles overlapping at method level. In Java Swing design, for example, it is common to have classes implementing both the *view* and *controller* roles. A menu item in a GUI, for instance, will be set as enabled/disabled if the command to be executed at the item's selection is enabled/disabled for the particular configuration of the application at the moment of selection. The implementation of the method relies on *controller* decisions, although the method is part of the *view* interface.

The refactoring of these concerns consist of overriding the method with an advice implementing the secondary role (which is the controller role in the GUI example).

The *Redirection layer* sort generalizes concerns that manifest themselves at the class level for a group of methods. The methods assume responsibility for the calls to an object, possibly perform additional actions like attaching responsibilities or modifying/casting the parameters of the calls, and then forward the calls to the object. Concerns of this sort are specific to implementations of the Decorator and Adapter pattern [7, 3] and are also discussed by Soares et al [8] when refactoring the *distribution* aspects for a web-based information system. The straightforward aspect solution consists of a pointcut and around advice to capture the calls and forward them [3]. However, this solution suffers from loss of flexibility because it is not dynamic [3, 8].

The two next sorts, *Add Variability* and *Expose context*, are very similar to two patterns proposed in [5]: *Worker object* and *Wormhole*. The concerns described in these patterns can be associated an object oriented idiom and a refactoring aspect mechanism, and they qualify as sorts. Our renaming aims to better show the intent of the concerns in this sort.

The associated refactoring of *Add Variability* is aimed at replacing the repetitive creation of *method objects* by putting it into an *around* advice, and triggering the advised method from the object's (execute) method.

Most of the sorts discussed up to now, are aimed in some way at providing a consistent behavior, so they can be generically included in a *behavioral* category of sorts. However, the object implementation idioms are different and this implies different guidelines for refactoring them. The next groups of sorts are addressed by aspect mechanisms that are relevant to static crosscutting and composition.

*Role superimposition* consists of extending the functionality of a class with crosscutting concerns by implementing multiple interfaces. *Support classes for role superimposition* follow a similar goal. Concerns in this sort are implemented by adding nested classes to the members of a hierarchy in order to enable the support for additional features. While the additional feature can be modularized in its own hierarchy, the relation enforced through nested classes results in code that crosscuts the enclosing classes. The introduction of supporting classes through inter-type declarations would enforce the same logical relationship provided by Java's nested classes mechanism, but would avoid the crosscutting.

*Policy enforcement* is an unsupported feature in object languages, but an object idiom can be recognized in the use of documentation and comments in the source code to define the policy. Two AspectJ language constructs, *declare warning* and *declare error*, support this sort of concerns by indicating at compile-time that a policy is violated. However, the main task of implementing this sort of concerns is still on the pointcuts definition to capture the access points.

A similar static mechanism, *exception softening*, can address *Exception propagation*. This allows to avoid the enforced rule of consistently propagating checked exceptions, by converting them into run-time exceptions.

The last three roles in Table 1 are currently unsupported by any aspect language mechanism. *Declare throws mechanism* was reported and requested in [8]. Instances of the next two are described in [7]. The object idiom in these two cases relies again on documentation and comments in the source code. However, an idiom can be observed for *dynamic behavior enforcement* in Java's use of *finalizers* to enforce the execution of specific code at the end of an object's lifecycle. These two sorts emphasize a consistent design discipline and object use, respectively.

## 5. Conclusions

The understanding and aspectization of crosscutting concerns proves to be difficult due to the lack of a coherent system to organize and describe such concerns. Aspect identification and refactoring is challenged by questions about (1) what symptoms or smells are specific to the crosscutting functionality at the implementation level, (2) what concerns are associated to these symptoms, and (3) which aspect mechanisms are adequate for refactoring and modularizing these concerns.

Contributions of this paper are as follows: we have proposed a classification system for crosscutting concerns based on *sorts*. In addition, we have proposed a first set of canonical sorts based on literature reviews and our practical experience with aspect refactoring [7, 1].

The classification of crosscutting concerns in sorts has a number of benefits: First, concern sorts help to develop a common language for consistently describing common situations of crosscutting functionality.

Second, the properties of a sort support the identification of crosscutting concerns in existing systems. On the one hand they help with the development of new aspect mining techniques by identifying the object-oriented idioms to search for. On the other hand they help developers understand results of aspect mining by mapping symptoms that were found in the legacy implementation to more abstract and generic concerns.

Sorts also allow to put together the theory and practice of refactoring, by showing risks and limitations in achieving aspect solutions for various concerns. These observations can provide feedback and contribute to the development of aspect languages that provide better support for refactoring.

## References

[1] A. van Deursen, M. Marin, and L. Moonen. A Systematic Aspect-Oriented Refactoring and Testing Strategy, and its Application to JHotDraw. Technical Report SEN-R0507, CWI, 2005.

[2] J. Hannemann, Murphy G.C., and Kiczales. G. Role-Based Refactoring of Crosscutting Concerns. In *Proc. of Int. Conf. on Aspect-Oriented Software Development*, 2005.

[3] J. Hannemann and G. Kiczales. Design Pattern Implementation in Java and AspectJ. In *Proc. of the 17th Annual ACM conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM Press, 2002.

[4] J. Kienzle and R. Guerraoui. AOP: Does It Make Sense? The Case of Concurrency and Failures. In *Proc. of the 16th European Conf. on Object-Oriented Programming*, pages 37–61. Springer-Verlag, 2002.

[5] R. Laddad. *AspectJ in Action - Practical Aspect Oriented Programming*. Manning Publications Co., 2003.

[6] M. Marin. An approach to aspect refactoring: Defining Crosscutting Concern Types. In *Int. Workshop on the Modeling and Analysis of Concerns in Software, ICSE*, 2005.

[7] M. Marin, A. van Deursen, and L. Moonen. Identifying Aspects using Fan-In Analysis. In *Proc. of the 11th Working Conf. on Reverse Engineering*. IEEE Computer Society Press, 2004.

[8] S. Soares, E. Laureano, and P. Borba. Implementing Distribution and Persistence Aspects with AspectJ. In *Proc. 17th Conf. on Object-oriented programming, systems, languages, and applications*. ACM Press, 2002.

[9] P. Tonella and M. Ceccato. Migrating Interface Implementation to Aspect Oriented Programming. In *Proc. Int. Conf. on Software Maintenance*. IEEE Computer Society, 2004.