

An Approach to Aspect Refactoring Based on Crosscutting Concern Types

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

We argue for the importance of organizing generic crosscutting concerns by distinctive properties and describing them as *types*. A *type*'s properties consist of a general intent, an implementation idiom criteria, and one (desired) aspect language mechanism to address the concerns within the specific *type*. We argue the usefulness of this approach for aspect refactoring, and in the areas of concern identification and aspect languages development.

1. INTRODUCTION

While refactoring legacy code to aspect oriented code (also known as *aspect refactoring*) one has to be aware of what are the elements to be refactored and what aspect solutions can replace them. This asks for coherent criteria to organize crosscutting concerns, so they can consistently be described and addressed. It also raises the question about what the right level of granularity is to make such an organization.

Related work in the area of aspect refactoring investigates the crosscutting functionality at different levels of granularity. The catalog of low-level refactorings proposed by Monteiro [11] describes code transformations from Java to AspectJ specific modularization units. Following a similar approach, Cole and Borba [2] introduce a set of programming laws for deriving AspectJ refactorings. These transformations are useful elements in the migration of crosscutting concerns to AspectJ; however, they are oblivious to the concerns to which they potentially apply.

Laddad [9, 8] covers a large range of refactorings, from logging to business rules implementation and design patterns. Design patterns are also discussed by Hannemann et al. [6, 7]. Many of the concerns discussed by these authors are complex and most of the time involve more than one crosscutting concern, which themselves can repeatedly occur in the implementation. A typical example is the Observer pattern where the concerns to be refactored are the *superimposed roles* of Subject and Observer, and the *con-*

sistent behavior of Subject changes notification enforced to the methods that alter the Subject object's state.

In this paper, we focus on organizing crosscutting concerns into *types*, i.e. generic descriptions of similar concerns that share a number of properties: (1) an intent, which can be described as a behavioral pattern or policy requirement, (2) a general implementation idiom in legacy (non-aspect oriented) code, and (3) a (desired) aspect language mechanism to modularize the concerns of the *type* into an aspect solution. A concrete concern is an *instance* of its *type*.

We choose to organize the concerns at the *aspect mechanism* level as this is the language design unit that allows to modularize concerns. For example, in terms of AspectJ, consistent behaviors can be expressed with a *pointcut and advice* mechanism, while roles superimposition can be addressed with *introductions*.

The objectives of this approach are to support modular refactoring, and to enable concern-based reasoning by providing a generic, formalized description of the crosscutting concerns on the one hand, and an associated solution consisting of an aspect language mechanism on the other hand. Types can be combined to express more complex situations, as the previous Observer example. Furthermore, a testing component could be defined for a type to ensure behavior conservation when refactoring its elements (instances).

The organization into types also has other benefits, such as a naming reference for describing typical instances of crosscutting functionality, and a system to assess the ability of the aspect languages to modularize crosscutting concerns.

Last but not least, the description of legacy implementation idioms associated with types provides useful information for the development of aspect mining techniques.

The next section gives an overview of the related work. It is followed by a definition of *crosscutting concern types* and an example of how these can be used to refactor concerns. The final sections propose a preliminary aspect catalog structure based on the discussed types.

The implementation language that we use for examples and discussions is AspectJ. However, we believe that many of the observations here are not restricted to this language only.

2. RELATED WORK

The approach we propose focuses on providing support for refactoring legacy systems to aspects and identifying crosscutting concerns in existing systems, so we will mainly discuss the related research in this area.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

MACS Workshop at ICSE 2005, St. Louis, Missouri, USA, May 16th, 2005
Copyright 2005 ACM ...\$5.00.

Refactoring	Description	Notes
General aspect refactorings [9, 8]	Aspect solutions to a number of crosscutting (demonstrative) examples	Large variety of examples; no specific categorization of the aspects, no automation, no specific testing component
Refactoring and testing strategy [3]	Refactoring and testing strategy for migration to aspects	Refactoring strategy including a testing component for behavior conservation, and its application to an open-source application; no automation, no specific categorization of the concerns
Role-based refactoring [6, 7]	Refactor design patterns	Design patterns refactoring; tool support for refactoring; no specific testing component
Catalog of code transformations [11]	Code transformations from Java to AspectJ	Do not address crosscutting concerns directly, but can be used to refactor parts of a crosscutting concern implementation or aspect code; do not discuss a testing component
AspectJ laws for refactoring [2]	Use programming laws to define behavior preserving transformations	Do not address crosscutting concerns directly, but can be used to refactor parts of a crosscutting concern implementation or aspect code; discuss a behavior conservation component

Table 1: Aspect-Refactoring approaches.

Table 1 orders a number of refactoring approaches from the coarse-grained (top) to the finer-grained ones. The top group of aspect refactorings, proposed by Laddad [9, 8], covers a number of examples of crosscutting functionality and the associated aspect solutions. Examples in this group include implementation of business rules, design patterns (*worker object creation*, *wormhole*), *extraction of method calls into aspects*, or *extraction of interface implementation*. However, as can be seen from these examples, these refactorings can be associated either with symptoms of a large variety of concern implementations (the last two), or with descriptions of general contexts into which the crosscutting functionality occurs (the first ones). There is no distinction between such situations¹, and also the specific refactorings can involve a variable number of crosscutting elements (such as roles or behaviors superimposed to classes or methods).

The refactoring and testing strategy that we proposed earlier in [3] describes a number of steps for guiding the identification and refactoring of crosscutting concerns. The strategy is applied to an open-source system and assesses the employed aspect language through a number of refactored concerns. Although some of them cannot be modularized by the existing aspect language features, reflection on these concerns has shown us that they all correspond to, or can be decomposed in, crosscutting concern *types*.

Hannemann et al. [6, 7] propose role-based refactorings for object-oriented design patterns [5]. The refactoring relies on a library of abstract descriptions of the patterns and their role elements, and instructions to refactor the pattern to an aspect solution. Tool support is provided for refactoring the code elements implementing the concern, after the user indicates the mapping between these elements and the abstract pattern description.

This approach is closest to ours as it looks for abstraction of problems involving crosscutting functionality. However, we believe that the refactoring of design patterns is typically complex and involves a number of crosscutting concerns, as earlier discussed for the Observer case. The one-step refactoring of the Observer will not ensure transparency for the

crosscutting concerns that occur in this context, and is vulnerable to deviations from a standard pattern implementation [6]. Furthermore, it is difficult to organize the crosscutting functionality at this level of complexity.

Both the AspectJ laws discussed in [2] and the code transformations catalogs presented in [11] are fine-grained refactorings. They describe code transformations that can occur as steps in the aspect refactoring of a crosscutting concern, but not the concern itself. These transformations can be from Java to AspectJ specific units (e.g., move method/field from class to inter-type) or AspectJ transformations (e.g., make aspect privileged, merge advices). Cole and Borba’s laws [2] also show an intuitive behavior conservation component of their transformations. Together with [3], these are the only explicit references to a testing component or strategy to ensure behavior preservation. However, it is acknowledged by all the others that such a component is a *sine qua non* of the aspect refactoring process.

3. TYPES OF CROSSCUTTING CONCERNS

3.1 The case for crosscutting concern types

Logging is very often mentioned as an example of a crosscutting concern. However, we claim that this is the case only if the logging functionality occurs as a secondary concern in a consistent, formalized context. That is, if the logging functionality is referred consistently by a number of elements (e.g., methods) that can be captured through a natural aspect language construct definition (e.g., a natural pointcut)².

The example in Figure 1 (taken from [9]) shows a consistent use of logging over the whole system for printing the names of the methods throwing exceptions. A typical implementation of the same concern in Java would consist of calling the logging method from all the methods throwing exceptions. The Java implementation idiom can be described as (*scattered*) *method calls*.

The same idiom and consistent behavior could be observed if instead of logging the exception, this would be wrapped into another exception to be thrown [10]. The refactoring

¹Work in [9, 1] mentions an aspect classification based on the phase of the software lifecycle at which the aspects occur: development and production aspects. This classification is not related to the discussion proposed here.

²A counter-example would be the occasional use of logging for local debugging

```

public aspect ExceptionLoggerAspect {
    pointcut exceptionLogMethods()
        : call(* *.*(..)) &&
        !within(ExceptionLoggerAspect);

    after() throwing(Throwable ex) : exceptionLogMethods() {
        Signature sig = thisJoinPointStaticPart.getSignature();
        log("Exception logger aspect ["
            + sig.getDeclaringType().getName() + "."
            + sig.getName() + "]);
    }
}

```

Figure 1: Log the exceptions throwing events in a system.

would be similar to the aspect solution for logging, consisting of applying a *pointcut and advice* mechanism.

The general intent of these concerns, as well as their idiom and aspect mechanism are also similar to the notification behavior in the Observer pattern previously discussed. This suggests that a general description and a reusable aspect solution can be associated to all these concerns, and makes the case for introducing generic *types* of crosscutting concerns to express these commonalities. In the next section we give a more formal definition of the *crosscutting concern types*.

3.2 Defining types

An important property of a *crosscutting concern type* in the definition we propose is the type's *atomicity*: there is only one (desired) aspect language mechanism to address the specific crosscutting functionality of a type, and an associated legacy implementation idiom.

To summarize, a *crosscutting concern type* is a generic description of a group of concerns that have three elements in common:

- a generic behavioral, design or policy requirement to describe the concern within a formalized, consistent context (e.g., role superimposition to modular units (classes), enforced consistent behavior, etc.),
- an associated legacy implementation idiom in a given (non-aspect oriented) language (e.g., interface implementations, method calls, etc.),
- an associated (desired) aspect language mechanism to support the modularization of the type's concerns (e.g., pointcut and advice, introduction, composition models).

Other examples of types, besides the *consistent behavior* discussed above, include *contract enforcement* (pre- and post- conditions), and *superimposed roles*. The latter's legacy implementation pattern can be described as interface (or group of methods that can be abstracted into an interface definition) implementation. The associated aspect language mechanism to refactor can be an AspectJ *introduction* or based on the HyperJ composition model. Two of the discussed types are shown in Table 2.

As is apparent from these examples, the implementation idioms and the aspect solutions are language specific. However, the purpose of adding these elements to the description of a type is precisely to imagine an aspect solution in a given language, and to assess the capability of the language to express crosscutting behavior.

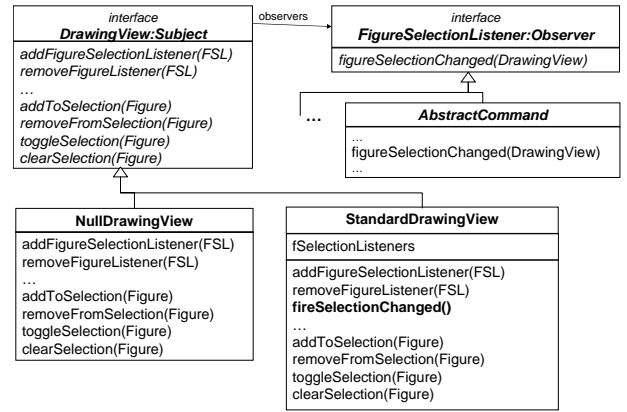


Figure 2: Observer pattern: Selection Listener.

4. REFACTORED USING TYPES

4.1 A case in point: the Observer pattern

Hannemann et al. [6] use an instance of the Observer pattern in the JHotDraw³ application to show their approach to aspectize design patterns. The application is a model framework for two-dimensional graphics, which also serves as a reference implementation of design patterns in Java. As we are currently developing an aspect-oriented version of the JHotDraw project [3, 4], we will illustrate the proposed type-based refactoring on the same Observer instance, *Selection Listener*, in JHotDraw.

The main participants in the pattern's implementation are shown in Figure 2. The *FigureSelectionListener* interface defines the Observer role as its primary concern. The interface is implemented by all classes interested in changes of the selection of figures in a drawing view. The *DrawingView* interface partially defines the Subject role by declaring two of the methods associated with the role: *add / removeFigureSelectionListener*. The Subject role is a secondary concern for the *DrawingView* interface. Two classes implement this interface and only one, *StandardDrawingView*, contains a non-empty implementation of the Subject role. Note that the implementation of the pattern exhibits a deviation from the standard one: the notification method specific to the Subject role, *fireSelectionChanged*, is declared (and implemented) only in the concrete Subject class.

4.2 Refactoring SelectionListener

The one step refactoring of the pattern to AspectJ [6] would result in a solution as partially shown in Figure 3. The aspect construct comprises both the Subject and Observer roles definition, and maintains a list of associations between each Subject and its Observer objects.

The type-based refactoring we propose distinguishes several crosscutting elements that occur in an implementation of the Observer pattern: *role superimposition*, applied twice, for each of the two roles, and *consistent behavior* to notify the observers of the changes in the subject object. These elements are shown in Figure 4. The *GenericRole* (empty) interface documents the crosscutting type of role superimposition. Specific roles, like Observer and Subject (*FigureSelectionSubject*) extend the interface.

³jhotdraw.org

Type	Behavioral pattern	Idiom	Aspect Mechanism	Instances
Consistent Behavior	Implement a consistent behavior for a number of method elements that can be captured by a natural pointcut	Method calls to the desired functionality	Pointcut and advice (language specific)	Logging exception throwing events in a system; Wrap service level exceptions of business services into application level exceptions; Notify changes of the Subject object state (in the context of the Observer pattern)
Role superimposition	Implement a specific (secondary) role	Interface implementation, or direct implementation of methods that could be abstracted into an interface definition	Introduction mechanisms (language specific)	Roles specific to design patterns: Observer, Command, Visitor, etc.

Table 2: Examples of crosscutting concern types.

The refactoring will further associate each type’s instance with an aspect module. Because this refactoring was done in the context of the larger AJHotDraw project⁴, some constraints are imposed by the adopted refactoring strategy [3]. The most important aims at ensuring behavior conservation after refactoring. As part of this strategy, a test suite is always executed on the original application and the refactored version, after each refactoring. This requires the conservation of the application interface and, as a consequence, the *FigureSelectionListener* interface is not declared in the aspect defining the Observer role, but only extends the generic role while its original declaration is not modified.

The context of the Observer pattern is modeled through a package definition. The type-based solution is the first step in the refactoring that moves each of the crosscutting elements to an appropriate aspect. Although more compact solutions could be abstracted, the benefits of the type-based approach can be discussed even at this stage:

- Each crosscutting element is addressed individually, which enhances a modular reasoning and a transparent migration of it. For each of these elements it is possible to evaluate the difficulties and the possibilities to be modularized into an aspect solution.
- Deviations from the pattern implementation in more than one participant can be addressed separately. In this respect, the aforementioned notifier method declaration and a number of other divergences discussed by [6] are good examples. Moreover, the definition of pointcuts to capture the calls to the notifier, as for many *consistent behavior* instances, is often difficult. In this example, the notification occurs only if specific conditions in the caller methods hold. The definition of the pointcut to capture the notification calls follows a less-intuitive logic occurring in the callers, where the *Figure.invalidate()* method is invoked if the same condition triggering the observers notification holds. Apparently, the pattern refactoring solution [6] notifies the observers independently of the condition in the caller. Although potentially harmless in this case, we argue that the conservation of behavior when refactoring should be an important issue, even if performed at a higher level of abstraction (conservation of intent). This brings us to the next point where the
- aspect solutions of the modular crosscutting types are mainly based on one aspect language mechanism. Fault

```

public aspect FigureSelectionUpdate {
    private WeakHashMap perSubjectObservers;

    protected interface Subject { }
    protected interface Observer { }

    declare parents: DrawingView extends Subject;
    declare parents: AbstractCommand implements Observer;
    //introductions for the other concrete observers

    public void addObserver(Subject subject,
        Observer observer) {...}
    public void removeObserver(Subject subject,
        Observer observer) {...}

    protected abstract void updateObserver(
        Subject subject, Observer observer);

    protected pointcut subjectChange(Subject s) : ...
    after(Subject subject): subjectChange(subject) {
        Iterator iter = getObservers(subject).iterator();
        while ( iter.hasNext() ) {
            updateObserver(
                subject, ((Observer)iter.next()));}
    }
}

```

Figure 3: Observer as a refactoring.

models defined for these mechanisms [3] can be associated with each refactored concern. This is less complex than a testing strategy ensuring behavior conservation for an one step refactoring of the Observer pattern.

- The more complex refactoring of the whole Observer pattern can be achieved from combination of these atomic refactorings.

The two types that we have identified in the pattern’s refactoring are also present in the one step solution, but they are not distinguished as such. In that case, the emphasis is on their combination in the context of the pattern. However, they are not specific to this context only, but can be encountered in various other contexts. *Role superimposition*, for instance, is very common in many design patterns implementation.

5. FURTHER ASPECT TYPES

The discussed types of crosscutting concerns could be part of an aspect catalog based on types. The entries in this catalog are generic concerns that will provide a common language for describing units of aspect refactoring, and from these ones, more complex modular refactorings. The catalog would include types like *Role superimposition*, *Consistent behavior*, *Contract enforcement*, and *Policy enforcement*

⁴sourceforge.net/projects/ajhotdraw/

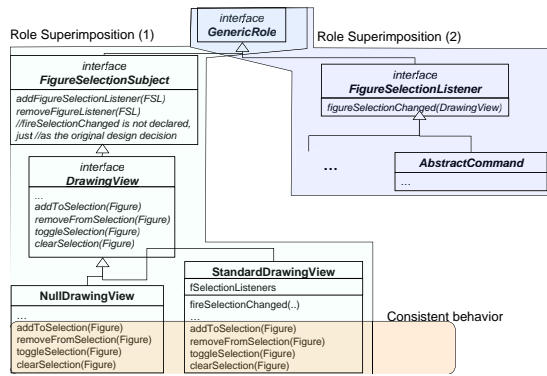


Figure 4: The concern types in Selection Listener.

(e.g., limit access to certain functionality) that are presently supported by various aspect languages.

Several other type candidates cannot be addressed by the existing aspect language features. These include:

- *Design enforcement.* The Bean aspect [1] is an instance of this type. The requirement to define a no-argument constructor for the Bean classes cannot be enforced through existing language mechanisms.
- *Dynamic behavior enforcement.* The Lifecycle concern discussed in [10] requires a start() operation to be called before any public method of a component, while a stop() operation terminates the object's use and should be the last call for a component's instance. Again this concern cannot be expressed by the present aspect-language features.

This second category can be used as a reference to extend existing languages with new mechanisms or to develop new model languages.

6. CONCLUSIONS

The contributions of this paper lie in the proposed criteria based on which to define *crosscutting concern types* to support the refactoring to aspects. These criteria are aimed at individualizing and describing groups of crosscutting concerns sharing common properties. The concerns are classified at the level of the aspect language mechanism that addresses their refactoring. This ensures transparency and flexibility of the refactoring.

We also identify the usefulness of our approach in the field of aspect mining, where the legacy idioms associated to *types* can be used to develop concern identification techniques. Furthermore, the proposed classification can help to develop new aspect language models and mechanisms to address the crosscutting functionality.

Future work consists of the identification of more types of crosscutting concerns and to provide a detailed description of these types in an aspect catalog. In parallel, we would like to use the type-based catalog to develop aspect refactorings with associated testing components. We also look at how mining techniques can be developed from the crosscutting concerns implementation idioms. A first step in this direction was made in [10].

7. REFERENCES

- [1] The AspectJ Team. *The AspectJ Programming Guide*. Palo Alto Research Center, 2003. Version 1.2.
- [2] L. Cole and P. Borba. Deriving Refactorings for AspectJ. In *Proc. Int. Conf. on Aspect-Oriented Software Development (AOSD)*, March 2005.
- [3] 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, Amsterdam, 2005.
- [4] A. van Deursen, M. Marin, and L. Moonen. AJHotDraw: A Showcase for Refactoring to Aspects. In *Proceedings of the Workshop on Linking Aspects and Evolution (LATE05)*. 4th International Conference on Aspect-Oriented Programming, 2005.
- [5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
- [6] J. Hannemann, Murphy G.C., and Kiczales. G. Role-Based Refactoring of Crosscutting Concerns. In *Proceedings of International Conference on Aspect-Oriented Software Development*, 2005.
- [7] J. Hannemann and G. Kiczales. Design Pattern Implementation in Java and AspectJ. In *Proceedings of the 17th Annual ACM conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 161–173. ACM Press, 2002.
- [8] R. Laddad. Aspect-Oriented Refactoring Series. www.theserverside.com, December 2003.
- [9] R. Laddad. *AspectJ in Action - Practical Aspect Oriented Programming*. Manning Publications Co., 2003.
- [10] M. Marin, A. van Deursen, and L. Moonen. Identifying Aspects using Fan-In Analysis. In *Proceedings of the 11th Working Conference on Reverse Engineering (WCRE2004)*. IEEE Computer Society Press, 2004.
- [11] M.P. Monteiro. Catalogue of refactorings for AspectJ. Technical Report UM-DI-GECS-200402, Universidade do Minho, 2004.