Role-Based Refactoring of Crosscutting Concerns

Jan Hannemann
University of British Columbia
201-2336 Main Mall
Vancouver, B.C. V6T 1Z4
jan@cs.ubc.ca

Gail C. Murphy
University of British Columbia
201-2336 Main Mall
Vancouver, B.C. V6T 1Z4
murphy@cs.ubc.ca

Gregor Kiczales
University of British Columbia
201-2336 Main Mall
Vancouver, B.C. V6T 1Z4
gregor@cs.ubc.ca

ABSTRACT

Improving the structure of code can help developers work with a software system more efficiently and more consistently. To aid developers in re-structuring the implementation of crosscutting concerns using aspect-oriented programming, we introduce a role-based refactoring approach and tool. Crosscutting concerns (CCCs) are described in terms of abstract roles, and instructions for refactoring crosscutting concerns are written in terms of those roles. To apply a refactoring, a developer maps a subset of the roles to concrete program elements; a tool can then help complete the mapping of roles to the existing program. Refactoring instructors are then applied to manipulate and modularize the concrete elements corresponding to the crosscutting concern. Evaluation of the prototype tool on a graphical editing framework suggests that the approach helps planning and executing complex CCC refactorings.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures – *patterns, information hiding, and languages;* D.3.3 [**Programming Languages**]: Language Constructs and Features – *patterns, classes and objects*

General Terms

Design, Languages.

Keywords

Design patterns, refactoring, and aspect-oriented programming.

1. INTRODUCTION

Aspect-oriented programming (AOP) [17] enables the modular implementation of crosscutting concerns. Refactoring [7, 24, 5] helps programmers improve improved code structure through behavior-preserving program transformations.

In this paper, we introduce a role-based refactoring approach to help programmers transform scattered implementations of crosscutting concerns (CCC) into a modular implementation in an AOP language. We use a role-based approach for two reasons. First, several authors have shown the value of roles in modularizing CCCs [20, 16, 9, 21, 27]. Second, we have shown in previous work [10] how a CCC that has been modularized conceptually by roles is amenable to modular implementation using AspectJ.

In the approach presented here, a refactoring is defined by describing the abstract roles of a CCC, and by providing refactoring instructions in terms of those roles. To invoke the refactoring, a developer maps the roles to concrete program elements in the scattered implementation. The refactoring instructions can then be used to operate on the program, converting it to a modular AOP implementation.

The approach is semi-automated, in that certain choices in the refactoring process need to be made by the developer. Options, tradeoffs and suggestions are presented by the tool in a form of dialogues we call *conversations*.

Our approach preserves intent, but in some cases not the exact behavior of the original code. We argue that these properties are inherent to a set of useful crosscutting concerns to modularize, a subset of the GoF design patterns [6].

We have implemented our approach as an Eclipse plug-in[1]. Our current plug-in supports Java and AspectJ, but we see nothing about our approach that prevents it from being used with other languages.

We begin the paper with a classification of the different kinds of aspect-oriented refactorings. We then introduce our role-based approach, describe the tool support we have built, and discuss its application to refactor design patterns in a graphical editing framework. We conclude with a discussion of issues surrounding our approach and a summary of our work.

2. REFACTORING SUPPORT FOR AOP

Refactoring support for AOP can be divided into three categories: aspect-aware OO refactorings, new refactorings for AOP constructs, and support for the refactoring of crosscutting concerns.

The refactoring focus—the element or elements that the refactoring targets—determines the refactoring approach, as outlined in Table 1. We use this refactoring space to clarify terminology and describe how our work relates to previous efforts.

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.
AOSD' 05, March 14-17, 2005, Chicago, Illinois, USA.
Copyright 2005 ACM 1-59593-043-4/05/03....$5.00.

[1] www.eclipse.org

Table 1: The AOP refactoring space

<table>
<tr><th colspan="2">Refactoring Focus</th><th>Approach</th></tr>
<tr><td rowspan="2">Single Program Element</td><td>OO Construct</td><td>Aspect-aware OO Refactorings</td></tr>
<tr><td>AOP Construct</td><td>AOP Refactorings</td></tr>
<tr><td colspan="2">Multiple, scattered program elements</td><td>Crosscutting Concern Refactorings</td></tr>
</table>

Existing refactorings often target modular implementations that are usually single program constructs, such as a field, method, class, advice or aspect. Although the focus is one program element, refactoring the element typically requires other parts of the program to be updated. When non-modular implementations are considered for refactoring, the target is a set of scattered program elements that belong together conceptually (i.e., they contribute to the implementation of the same concern).

```
public class Account {
  ...
  public void deduct(int amount)
   throws InsufficientFundsException {
    acquireLock(this);
    doIt(amount);
    releaseLock(this);
  }
  private void doIt(int amount)
   throws InsufficientFundsException {
    if (amount > balance) {
      throw new InsufficientFundsException(...);
    } else {
      balance = balance - amount;
      transactions++;
    }
  }
}

public aspect TransactionLogger {
  ...
  pointcut deduction(Account acct, int amount):
    call(void Account.doIt(int)) &&
    target(acct) &&
    args(amount);

  after (Account acct, int amount) returning:
   deduction(acct, amount) {
    log(amount + " deducted from " + acct.getId());
  }
}
```

Figure 1: Source code fragment demonstrating logging of deductions in a banking system

Figure 1 introduces an example in AspectJ [18] that we use to motivate and illustrate these different types of refactorings. It shows a part of an account class for a banking system that focuses on handling deductions. In the method `deduct(int)`, a lock is acquired (to prevent race conditions), then the actual deduction is executed (in method `doIt(int)`), and finally the lock is released. If the balance is not sufficiently high, an exception is raised. Further, an aspect logs all successful transactions; Figure 1 shows the logging of successful deductions.

2.1 Aspect-Aware OO Refactorings

Known OO refactorings must be adjusted to account for new AOP constructs. For instance, applying a Rename Method refactoring to the (poorly named) `doIt(int)` method requires updates to references of that construct from AOP constructs, such as the `deduction(..)` pointcut. Applying a refactoring to inline the `doIt(int)` method is another example where an OO refactoring needs to become aspect-aware, as the joint points [13, 22] identified by the pointcut cease to exist. Making OO refactorings aspect-aware is the focus of several current research projects, for example [8, 14, 29].

2.2 Aspect-Oriented Refactorings

New refactorings are also needed to transform AOP constructs; many of these refactoring may parallel existing OO refactorings, such as renaming or inlining a pointcut and advice body. With respect to the type of refactorings that can be applied to them, pointcuts behave very similar to methods and aspects to classes. It is straightforward to envision the meaning of refactorings such as Add Parameter, Pull Up Method, or Push Down Method [5] to a pointcut declaration. Similarly, equivalents of Collapse Hierarchy, Extract Super/Subclass, or Move Class may be modified for aspects.

The new programming constructs in aspect-oriented programming languages also allow for a set of new refactorings, such as the merging or splitting of advice and/or pointcuts. For example, imagine a similar pair of pointcut and advice in the banking system for logging additions of money to an account. The pointcuts, the advice, or both could be merged with the ones for logging deductions. Several researchers have proposed new refactorings for AOP constructs, for example [23, 14].

2.3 Refactorings of Crosscutting Concerns

Refactorings are also needed for crosscutting concerns. As a simple example, consider an object-oriented system that logs certain method calls. In an object-oriented system, the calls to the logging facilities will likely be scattered across multiple classes and methods. Replacing all of these non-modularized with a pointcut and advice is a CCC refactoring. In such refactorings, the multiple program elements comprising the CCC and their individual transformations are considered together, instead of handling each element (method call in the logging example) separately.

For a more interesting example, consider an object-oriented instance of the Observer design pattern [6], which is by its nature non-modularized (as is the case for many OO design patterns). We have shown in previous work that many design patterns can be modularly implemented in AspectJ [10]. Moving from an object-oriented to an aspect-oriented implementation is a CCC refactoring. The program elements comprising the CCC have certain functions and relationships that make it difficult to consider them separately. In a CCC refactoring, the individual transformations for these program elements are planned and executed together. The individual transformations involved are all either (aspect-aware) OO or AO refactorings.

To illustrate this, consider the steps needed to modularize an instance of the Observer pattern in the JHotDraw framework [15]. Figure 2 shows the structure of this particular pattern instance, and the methods comprising the concern. Relevant code elements appear in all observers, and all subjects, plus in client code (not shown) that sets up the subject-observer relationships.

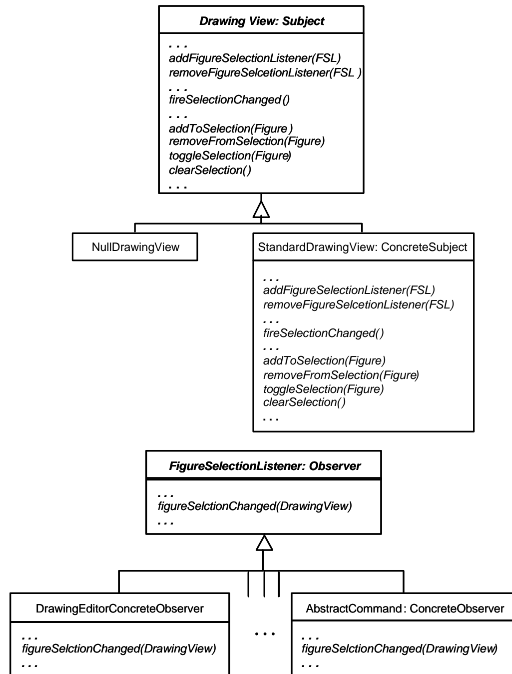

Figure 2: The structure of an (object-oriented) instance of the Observer pattern in the JHotDraw framework.

In this case, we have `DrawingView` acting as the subject. It has methods to add and remove *Observer*s (here: `FigureSelectionListeners`), and to notify observers of a change of interest. Further, we have a number of methods that can trigger notification. `DrawingView` is an interface; concrete subtypes implement these methods. The interface is implemented by two types directly in the framework, and is also a primary extension points of the framework. Concrete applications built upon the framework are thus likely to have additional classes containing subject code.

`FigureSelectionListener` acts as an *Observer* and declares a method to update itself if a change of interest occurs. This interface is implemented by a total of 31 types in the framework plus 14 different anonymous types.

This pattern crosscuts a large number of modules, tangling the scattered concern implementations with other code, making changes complex. If we consider the final, modularized, implementation of the Observer pattern CCC as an aspect, it takes a rather long list of changes to move the code from the scattered to the modularized aspect form:

1. Create an aspect to contain the pattern instance (named `FigureSelectionUpdate` in light of the previous example)
   a. Add a field (of type `HashMap`) to the aspect to store the Observers of each Subject
   b. Define empty interfaces Subject and Observer in the aspect to provide some basic internal typing
   c. Use `declare parents` constructs to assign the Subject and Observer interfaces to the appropriate types
   d. Implement a pointcut `subjectChange(Subject)` capturing all changes of interest to subjects
   e. Implement a method `update(Subject, Observer)` that updates the observer given a change in the subject
   f. Implement the update logic: after `subjectChange`, call `update(Subject, Observer)` for all observers of the subject
2. Adjust the original subject interface (`DrawingView`)[2]
   a. Remove method `addFigureSelectionListener(..)`
   b. Remove method `removeFigureSelectionListener(..)`
   c. Remove method `fireSelectionChanged()`
3. Adjust the concrete subjects (`StandardDrawingView`, `NullDrawingView`, all framework extensions)
   a. Remove method `addFigureSelectionListener(..)`
   b. Remove method `removeFigureSelectionListener(..)`
   c. Remove method `fireSelectionChanged()`
   d. Remove the `figureSelectionListeners` field or its equivalent
   e. Locate and remove calls to `fireSelectionChanged()` from all methods
4. Remove the original observer interface (`FigureSelectionListener`)
5. Adjust the concrete observers (`AbstractCommand`, `DrawingEditor`, and all other types)
   a. Remove the method `figureSelectionChanged(DrawingView)`
6. Add appropriate code to `FigureSelectionUpdate.update(Subject, Observer)` to reflect the deleted code
7. Adjust all client code
   a. Locate all calls to `addFigureSelectionListener(..)` targeting a `DrawingView` or a subtype and replace it with an appropriate call to `FigureSelectionUpdate.addObserver(Subject, Observer)`
   b. Locate all calls to `removeFigureSelectionListener(..)` targeting a `DrawingView` or a subtype and replace it with an appropriate call to `FigureSelectionUpdate.removeObserver(Subject, Observer)`

There are two things to note about this conversion: first, this is a rather long list of changes and over 50 types are affected directly or indirectly. Second, between the individual refactoring steps, the

[2] This interface can not be entirely removed, as it contains other code that is not related to the pattern

system is left in an inconsistent state, suggesting that treating the entire list as one refactoring as one is preferable.

If we were to do the same transformation for a different instance of the Observer pattern, we would have to modify the involved elements in the same principal way, yet we would target a different set of program elements, with entirely different names. Ideally, we would capture the common steps and only slightly adjust the "modularize OO Observer pattern" refactoring each time it is applied to a new instance of the pattern. This is the motivation for CCC refactoring.

Our approach to CCC refactoring is *role-based*. The use of roles allows a CCC refactoring to be defined separately from the concrete systems to which it may be applied. For CCC descriptions, roles help identify and distinguish between principal elements of the concern, for example the *Subject* type or the *update(..)* method in the Observer CCC. These role elements abstractly define the structure of the CCC without tying it to a concrete implementation. To apply a CCC refactoring, a developer must map the role elements comprising the abstract description of the CCC to the concrete program elements implementing the CCC. The refactoring instructions are also defined in terms of these role elements, and the refactored code is obtained by applying these instructions to the mapped program elements.

We are not aware of any work on role-based refactoring support for CCCs. Fowler touches on the issue of complex OO refactorings with a description of what he calls "big refactorings" [5], and presents the rough mechanics (similar to the list of changes above) of four examples. Shepherd [26] proposes Ophir, an automated tool for mining and extracting aspects. Based on clone detection in program dependency graphs [4], their approach does not model CCCs, nor utilize the CCCs' internal structure. Laddad [19] describes a number of OO-to-AOP refactorings that extract scattered implementations into aspects; these descriptions are not supported by a tool. Robillard's concern graphs [25] are a way to capture the structure of non-modularized crosscutting concerns, but they do not provide any refactoring support. In [11], Hannemann et. al. use concern graphs to plan refactorings, and discuss the role of developer interaction in providing tool support for refactoring scattered code to aspects. However, concern graphs do not include any abstraction to differentiate between individual concrete program elements, limiting the description of complex refactorings.

2.4 Properties of AOP Refactorings

We have found that AOP refactorings, and especially CCC refactorings, have two interesting properties. One, with this kind of refactoring there is sometimes a trade-off between behavior preservation and preservation of intent. Two, choices in the application of the refactoring and space of possible aspect-oriented implementations raise the need for user interaction as part of the refactoring process. The more complex the refactoring, the more pronounced are these two issues. Yet, even the rather simple inlining of the `doIt(int)` method in the example shown in figure 1 can illustrates both points.

To preserve the original behavior, we could define the logging to happen before the next program statement is executed (i.e.: before the call to `releaseLock(..)`), or we could decide the logging should happen after the body of the original, inlined method (where the `transactions` field gets incremented). Both approaches fail to capture the original intent of the pointcut and make it less readable and self-explanatory.

To maintain readability of our pointcut, we could specify that logging should take place after the balance field is modified in method `deduct(int)`. This would require accepting a minor behavior variation (which is against the fundamental principle of refactoring), but the pointcut would remain readable, and the intent of the pointcut remains intact.

This also illustrates the issue of required developer interaction: the decision to choose one alternative over another is hardly automatable. While it is conceivable that a tool can just pick one possibility and provide reasoning for later review (e.g., using JSR 175 metadata annotations), a complex refactoring may require multiple choices, each of which can influence subsequent refactoring steps.

3. ROLE-BASED REFACTORING OF CROSSCUTTING CONCERNS

Our approach helps a developer transform a scattered implementation of a CCC into an equivalent[3], but modular AOP implementation. We describe both the CCC and the refactoring abstractly in terms of roles. Refactorings are executed by applying the refactoring instructions to the program elements that the roles are mapped to, as outlined in figure 3. Dependent code elements are changed accordingly, so the refactoring can indirectly affect non-mapped elements as well.

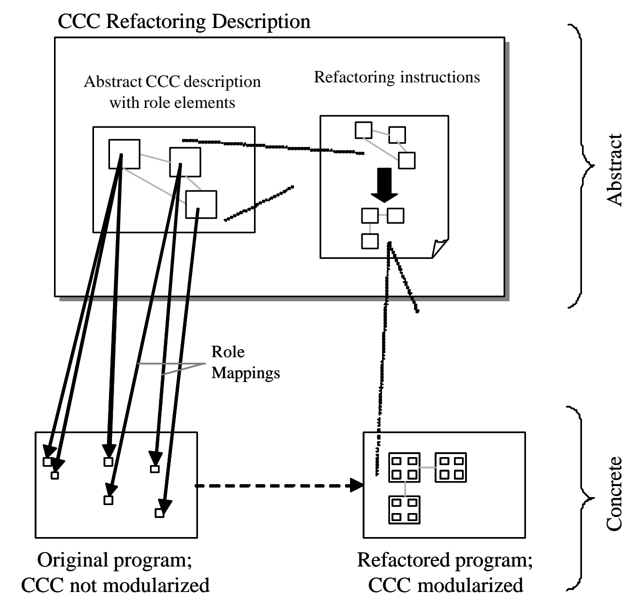

Figure 3: CCC refactorings as code transformations using role abstractions.

3.1 Workflow

Performing a CCC refactoring involves several steps. We describe these steps using the `FigureSelectionUpdate` example from the previous section.

---

[3] in terms of preserving intent

1. **Selecting a CCC refactoring:** The developer chooses an appropriate refactoring from a library of CCC refactorings. The refactoring includes an abstract description of the CCC it targets, and a set of instructions to produce a modular AOP implementation of the refactoring. These refactorings can be user-defined, and, due to their abstract nature, reused for other instances of the same CCC. In the FigureSelectionUpdate example, the developer chooses the Observer design pattern CCC refactoring
2. **Stating a mapping:** The refactoring tool helps the developer map role elements comprising the CCC description to program elements of the scattered implementation. In the example, the developer might map the *Subject* role type to `DrawingView` and the *Observer* role type to the concrete type `FigureSelectionListener`. The tool would then suggest further mappings that the developer can accept or reject.
3. **Planning the refactoring:** Because a CCC refactoring involves modifying several parts of a codebase, there are often choices that arise in the refactoring. The refactoring tool uses impact analysis to identify choices[4] in the refactoring process, and to provide the developer with associated tradeoffs. For example, the tool would warn the developer if the newly created aspect's name collides with another entity in the program, or if one of the introduced methods is accidentally matched by an existing pointcut. The developer decides how to resolve these cases.
4. **Execution:** Once the refactoring has been planned, the tool transforms the code according to the refactoring instructions, incorporating the decisions made in the planning step. The execution of the refactoring may result in the creation of new program elements, such as a new aspect to contain the modularized Observer pattern, as well as changes to the existing code, such as the replacement of object methods with aspect methods and the removal of obsolete interfaces.

3.2 Describing Crosscutting Concerns

To enable the refactorings to be described abstractly, we differentiate between concrete and abstract CCCs. A *concrete* CCC is the actual implementation of the concern, while an *abstract* CCC description generalizes and captures the structure of concrete CCCs. For example, in the JHotDraw framework, there are multiple cases where the Observer pattern has been used. These are multiple concrete scattered Observer CCCs. All of them have a similar structure, which is captured in the abstract Observer CCC.

Each CCC refactoring in the library includes an abstract description of the CCC it targets. The abstract CCC is described in terms of a set of role elements and a set of relations between role elements. Each role element abstracts part of the functionality of the concern. We differentiate between three kinds of role elements: role types, role methods and role fields. Table 2 depicts the possible relations of interest between the role elements. These role elements and their relations are the core of the model; the model can be extended with additional relationships and properties, such as methods modifying fields, types extending types, or types being abstract or concrete. The relations shown in Table 2 have been sufficient to represent the set of design pattern CCCs we have studied to date.

[4] or potential adverse effects of changed program elements on the rest of the system

Table 2: Role Elements and their Relationships

	Possible Relationship with other Role Elements		
Role Element	**Type**	**Method**	**Field**
Type	extends	contains	contains
Method	returns, hasArgument	calls, overrides	-
Field	hasType, aggregates	-	-

For instance, the abstract CCC description of the Observer design pattern includes the role types *Observer* and *Subject*, and the role methods *attach*, *detach* and *notify* on the *Subject*, and *update* on the *Observer*. Role fields are *observers* on *Subject* and (optionally) *subject* on *Observer*. This structure is shown in figure 4. Note that the distinction between Subject and ConcreteSubject (as in the design pattern description) is an implementation issue and not a conceptual one; hence it is not reflected in the CCC description.

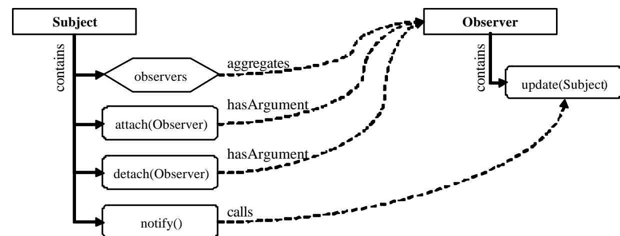

Figure 4: Role Elements in the Observer CCC. Role types are represented as squares, role methods as rounded squares and role fields as hexagons. The structure reflects the *push* variant of the pattern as outlined in [6].

Given the spectrum of implementation variants for CCCs, an important question is how to deal with potentially varying concern structure, such as known implementation variants for design patterns. Section 5 discusses this issue in detail.

3.3 Mapping Role Elements to Code

Applying a CCC refactoring requires a mapping from the abstract CCC description to program elements in a code base that comprise a concrete CCC implementation.

This mapping is aided by the approach. Based on initial partial mapping information provided by the developer, a comparison of the structure of the abstract CCC and a static analysis of the type hierarchy and call graph structure of the target software can be used to suggest additional mappings. For instance, if a developer has

mapped the *Subject.attach(Observer)* role method onto `DrawingView.addFigureSelectionListener( FigureSelectionListener)`, a tool can straightforwardly infer a mapping of the *Subject* role type to `DrawingView` (via the "contains" relationship between *Subject* and *attach*) and of the *Observer* role type to `FigureSelectionListener` (via the "argument" relationship between *attach* and *Observer*). Figure 5 shows the initial and derived mappings. This process can be applied iteratively using derived mappings as the basis for further inferences.

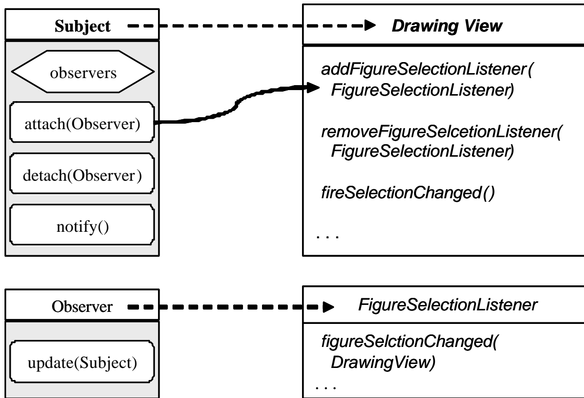

Figure 5: Based on an initial mapping (solid arrow), further mappings can be inferred (dashed arrows).

A comparison of the structure in the abstract CCC and the program structure can also reveal potentially incorrect mappings, and suggest alternative mappings that match the abstract CCC structure more closely. For example, it can be detected whether the concrete methods that a role method is mapped to have the wrong argument or enclosing type, or whether the implementation is missing parts of the structure of the CCC, such as an explicit interface. In both cases, the developer is presented with the current mapping and the mapped elements that do not match the structure of the CCC description. Potential alternatives are evaluated based on the fit of the CCC description and the structure of the mapped elements. Structural information inherent in the CCC description provides an initial ranking (the mapping with the fewest deviations has the best fit – all role element relationships are considered equal for this purpose). Note that the algorithms for completing and checking partial mappings are not tied to a specific CCC, but rather work on any abstract CCCs expressed in terms of their roles..

To further facilitate the mapping process, it is also possible to augment the abstract CCC description with non-structural information about the role elements, such as lexical or semantic information, as described in sections 4 and 5.

In summary, the structure of the concern description can aid in finding scattered program elements directly corresponding to role elements. Remaining scattered locations of the CCC, such as call sites for a mapped method need not be explicitly identified in this phase and will be automatically updated when the code is transformed (see Code Manipulation, below).

3.4 Describing Refactorings for CCCs

To execute a CCC refactoring, we need a set of instructions to transform the code base and modularize the scattered concern. The refactoring instructions are described in terms of the role elements of the abstract CCC. Each individual refactoring instruction is either an aspect-aware version of an existing object-oriented refactoring [5, 24] or an aspect-oriented refactoring. A refactoring step applies to all program elements to which the role elements are mapped.

To aid our description of the refactoring instructions, we introduce notation to express the mapping between program elements in the software system to abstract role elements. We use $m : RE \rightarrow P$ to refer to the mappings *m* between the set of role elements *RE* and the set of program elements comprising the target program *P*.

The set of concrete program elements that a particular role element $re \in RE$ is mapped to, is expressed as $m(re) = \{pe_i \in P \mid re \text{ is mapped to } pe_i\}$.

To illustrate the kinds of refactoring steps for a CCC, we describe the steps involved in transforming an object-oriented instance of the Observer pattern (shown in [10]) into an aspect-oriented version. The refactoring instructions we use for the example assume a library aspect that abstracts commonalities of the Observer pattern from the aforementioned work.

Figure 6 shows the structure of this abstract library aspect. The empty interfaces Subject and Observer are used internally to provide a form of typing for the pattern methods, pointcuts and advice. They do not define behavior, which is instead provided by the aspect. The library aspect contains most of the pattern functionality, including facilities for the addition and removal of observers per subject, and the notification logic that triggers observer updates when a change on interest occurs. For each concrete instance of the pattern, a developer only need to subclass this aspect, specify which types act as Subjects and Observers, concretize the `subjectChange` pointcut to specify what constitutes a change of interest, and override the update method to define the update behavior.

```
public abstract aspect ObserverProtocol {
    ...
    protected interface Subject  { }
    protected interface Observer { }

    protected List getObservers(Subject subject) {...}

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

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

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

Figure 6: The abstract library aspect for the observer pattern

The refactoring steps for Observer are described below. For brevity, instructions for removal of program elements rendered obsolete by the modifications are omitted (empty interfaces, unused variables, etc.). The aspect-oriented refactorings named are described in the appendix.

1. Create a new aspect to modularize the CCC and assign it a name selected by the developer. In the case of the Observer CCC, we can extend an existing library aspect implementation. We call the new aspect `FigureSelectionUpdate`.
2. For the mapped role types, specify which internal interface they should implement to represent the role types in the modular implementation. This is realized by the *Add Internal Interface* refactoring. Here, we specify which types act as Subjects and which act as Observers:

   $\forall pe_i \in m(Subject)$: *Add Internal Interface*: `Subject` to `FigureSelectionUpdate`

   $\forall pe_i \in m(Observer)$: *Add Internal Interface*: `Observer` to `FigureSelectionUpdate`
3. For mapped role methods, replace methods with aspect methods. For the Observer CCC, replace the existing methods for adding, removing, and updating observers with the provided aspect methods:

   $\forall pe_i \in m(attach)$: *Replace Object Method* pe_i *with Aspect Method* `FigureSelectionUpdate.addObserver( Subject, Observer)`

   $\forall pe_i \in m(detach)$: *Replace Object Method* pe_i *with Aspect Method* `FigureSelectionUpdate.removeObserver( Subject, Observer)`

   $\forall pe_i \in m(update)$: *Replace Object Method* pe_i *with Aspect Method* `FigureSelectionUpdate.updateObserver( Subject, Observer)`
4. Consider and update other parts of the abstract CCC structure, such as the call structure. In the Observer case, replace the update logic. Changes of interests occur within methods on the Subjects that call the *notify* role method. The implementing method(s) (and explicit calls to it) is/are replaced by a pointcut and advice code:

   $\forall pe_i \in m(notify)$: *Replace Method Call to* pe_i *with Pointcut* `FigureSelectionUpdate.subjectChange(..)` *and Advice*.

3.5 Impact Analysis and Conversations

At various points in the workflow of role-based refactoring for CCCs, we utilize static program analysis to assess the impact of a possible change to the code base. Based on the phase in the workflow, different kinds of analyses are employed. The impact analysis employed during the refactoring phase checks for a number of decision points that require developer interaction to resolve.

Whenever a check performed by impact analysis discovers a decision to be made by the programmer, it will initiate a conversation with the developer, presenting the associated information to allow a decision to be reached. Concretely, the developer is always presented with four pieces of information: the situation, the alternatives, the associated tradeoffs, and a recommendation.

In the refactoring planning phase, impact analysis helps identify proposed changes that can have an undesired impact on other parts of the system. Simple examples include the detection of naming collisions or unintentional pointcut matches. For the former, the developer would be presented with the new and old program element in question (the situation), the choice to rename either one (alternatives), the impacted references in the rest of the program in each case (tradeoffs), and the suggestion to rename the one with fewer dependencies (recommendation). An extensive set of potential problems associated with altering program elements in an aspect-oriented environment during the refactoring can be found in [12]. Note that the system is not changed in the planning phase, and the analyses are instead performed the on proposed program's AST.

Although most decision points can be sufficiently analyzed using existing static analysis techniques, more complicated checks may require more sophisticated analysis methods. For instance, aspect interference can be problematic: if a newly introduced pointcut matches a joinpoint that is also matched by an existing pointcut, aspect/advice ordering may have an effect on program behavior. Unless precedence is declared, the repercussions on the system are difficult to determine without asking the developer. Similarly, introducing additional interfaces via the declare parents construct can change the dynamic behavior of the system. In general, potential problems like these are often easy to detect, but their actual implication on the system can be hard to determine with static analysis alone [12]. In such cases, it is often easier to present the case as a decision point to the developer.

Although user interaction during a refactoring is not restricted to CCC refactorings, a developer might have to make a number of choices over the course of a complex refactoring,. Since some choices can influence later decisions, the communication flow between developer and supporting technology can be seen a dialogue[5]. We call this form of interaction a *conversation*.

3.6 Code Manipulation

Once the refactoring has been planned and all of the associated decision points in its realization have been resolved, a supporting tool can perform the code manipulations. It follows the refactoring description, performing the instructions on the mapped concrete program elements. Other, non-mapped, program elements that reference the changing portions of the program will be updated accordingly.

4. THE REFACTORING TOOL

We have developed a tool for role-based refactoring of CCCs and realized it as a plugin for the Eclipse IDE. Figure 7 shows an overview of architecture of the tool.

The tool relies on three forms of representation of the program that are derived from the source, all via existing tools. The Eclipse parser produces two models of the program sources, an AST and the Java Development Tools (JDT) model. We mostly rely on the JDT model in our tool as it provides a suitable and easily accessible

[5] In the traditional, non-UI sense.

interface to obtain information about program elements. The AST representation is used only within the refactoring engine for certain searches. The JavaDB module utilizes the AST information to generate a database of program facts that complements the other models[6]. The AutoMapper uses this database to find methods based on their argument or return type(s), and to determine call relationships between methods.

The AutoMapper module facilitates the mapping of role elements onto concrete program elements. Based on one or more initial mappings provided by the developer, it attempts to match the CCC structure with the structure of the target program. The program structure used comprises hierarchical relationships derived from the ASTs and method call relationships as provided by the JavaDB program facts database. This structural analysis generates candidate program elements for unmapped roles. To rank multiple candidates for a single role, we use lexical information about the role elements stored in the abstract CCC descriptions. Lexical information is a set of name fragments (currently just substrings) that are likely to be used for concrete program elements playing a particular role. For example, it is more likely that a concrete implementation of the role method *attach(Observer)* has a name containing either "add" or "attach", than "remove", or "detach". If the developer accepts one or more suggestions, the analysis starts over, taking into account the new mappings.

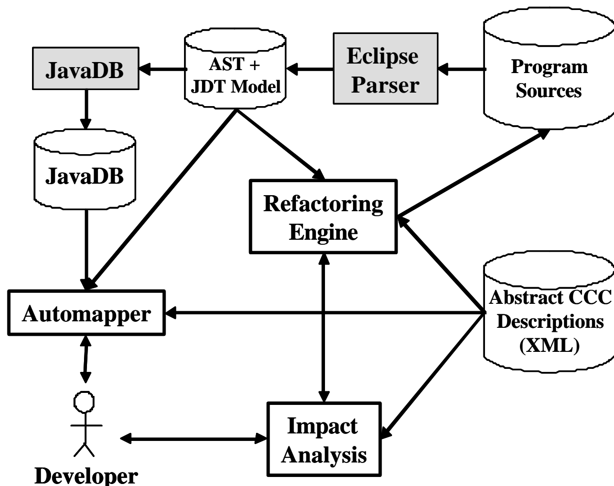

Figure 7: The CCC refactoring tool architecture. Existing software used is shown in gray.

The Refactoring Engine is responsible for planning and executing the CCC refactorings. Each CCC refactoring is currently realized as a set of hard coded calls to a number of elementary aspect-oriented refactorings (see appendix). These elementary refactorings are implemented as Eclipse refactorings; they are extensions of the abstract `Refactoring` Eclipse class. When planning a refactoring, the Refactoring Engine has each instruction checked by the Impact Analysis module, which can reveal decision points to present to the developer. Once all potential decision points have been resolved, the source code is changed to reflect the refactoring instructions. We currently realize these changes as buffer manipulations of the affected compilation units, such as text changes to the Java files. We chose to use the buffer manipulation approach because of the current flux in the components we use, specifically AJDT[7] and the Eclipse refactoring framework. The Refactoring Engine executes all necessary code transformations to create the target concern structure. This includes creating aspects, modifying or removing original program elements, and updating references to them. In a few cases, the developer might have to adjust the created code slightly, for example when an object method is replaced with an aspect method as this may affect scoping of statements within the original method. These adjustments usually have a granularity of statements rather than types or methods.

The Impact Analysis module ensures that a refactoring does not adversely affect the rest of the system. Currently, for each elementary refactoring, the module checks a number of preconditions, such as ensuring that the targeted method for *Replace Method with Intertype Declaration* is not protected. If a new program element is created, the module checks the name space to avoid conflicts with existing elements. The module generates the appropriate conversations so that identified decision points can be resolved by the developer.

5. JHOTDRAW STUDY

To ensure that our approach and tool are adequate to refactor scattered CCCs in a non-trivial code base (over 240 types in 15 KLOC) not written by us, used our tool to refactor six instances of three different design patterns in the open source JHotDraw graphical editor framework [15]. In addition to Observer discussed above, we investigated instances of Singleton and Template Method [6]. In this sample, all of the CCC refactorings were behavior preserving. However, as we described earlier (Section 2), behavior preservation may not always be possible.

5.1 Singleton

Singleton is a simple pattern. Our description of the pattern as an abstract CCC was based on our previous work [10]. It consists of the role type *Singleton* and the role method *instance()*, as shown in Figure 8. The figure also shows concrete program elements mapped to the role elements, such as the `Clipboard` class mapping to the Singleton role type.

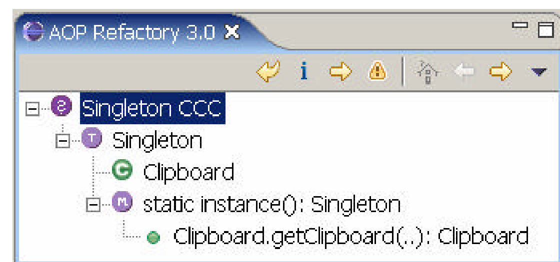

Figure 8: The Singleton CCC structure with a sample mapping

The CCC's refactoring instructions were:

1. Generate an aspect to modularize the pattern
2. Change the access modifier of the singleton's constructors to public

---

[6] JavaDB is part of the FEAT tool [4].

[7] www.eclipse.org/ajdt/

3. Replace calls to instance() with calls to the constructors

We refactored two instances of this pattern in JHotDraw. We found these instances using a simple text search over the program sources for the word "Singleton".

The refactored instances use different variants of the pattern. One uses a protected non-parameterized constructor and an *instance()* method, as outlined in [6]. The second features a non-protected, parameterized constructor. We determined that this variant represented a singleton only when we found a comment explaining it was a Singleton in the source code documentation. We assume that the constructor was not protected to ensure that proper arguments were passed upon generation of the singleton instance. The CCC library refactoring instructions did cover this case and did not have to be adjusted.

Since the structure of the CCC is minimal, we did not use automapping in this case. The type names of the original singletons did not indicate their singleton nature, so the lexical rules did not match in these two cases[8]. The refactorings of the two instances changed three and four types, respectively, in the JHotDraw code base as it was transformed to the AOP version.

5.2 Template Method

The Template Method CCC has the role types *AbstractClass* and *ConcreteClass*, plus the role method *templateMethod()*. Figure 9 shows the CCC structure, with the mappings from a concrete instance.

Again we scanned the sources to find an instance of the pattern and to provide an initial partial mapping. Starting with a mapping of *abstractClass* to a concrete type, the automapping properly identified the four immediate subtypes as candidates for the *concreteClass* role. We mapped the role method *templateMethod* by hand as the defining characteristic (a concrete method calling non-concrete methods) was too complex to be captured by our automapping rules.

The library CCC refactoring involved the following instructions:

1. Create an aspect to modularize the pattern
2. For all concrete methods that the role method templateMethod() is mapped to: *Replace Method with Intertype Declaration*
3. For all fields on AbstractClass: *Replace Field with Intertype Declaration*
4. If AbstractClass has no remaining members, change it from abstract class to an interface.
5. If AbstractClass changed to an interface: For all types that the role type ConcreteClass is mapped to: update references to AbstractClass

In the occurrence we investigated, the concrete type playing the *abstractClass* role had four fields and provided implementations for 31 methods, 10 of which are template implementations (i.e., they refer to abstract methods that subtypes implement). We found the refactoring to be straightforward, although we required developer interaction in two cases: it is not possible in AspectJ to use inter-type declarations to declare a protected method or static fields using that mechanism, so the developer needs to adjust the code accordingly. For the two cases, a total of 5 and 4 types were changed for this reason, respectively.

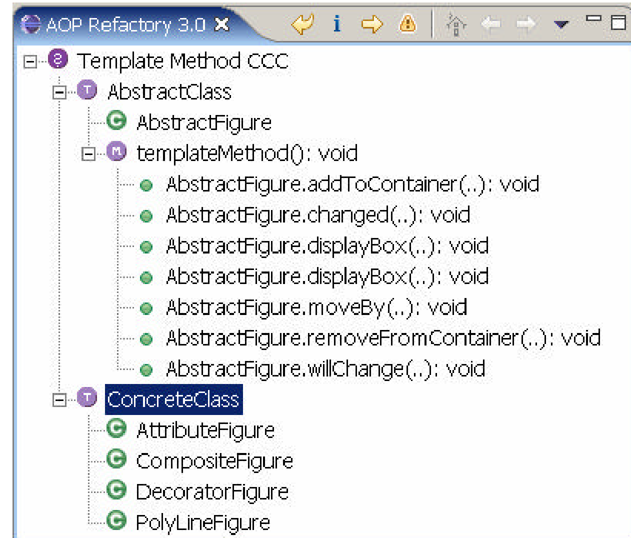

Figure 9: The Template Method CCC with a sample mapping

5.3 Observer

The Observer case was more complex, as the pattern has considerably more internal structure. The CCC comprises two role types, five role methods and one role field, as shown in figure 10.

All four instances of the pattern we investigated deviated in one or more ways from the description in the abstract CCC. We address the deviations in section 6.1. The necessary refactoring instructions for this pattern were presented in section 3.4.

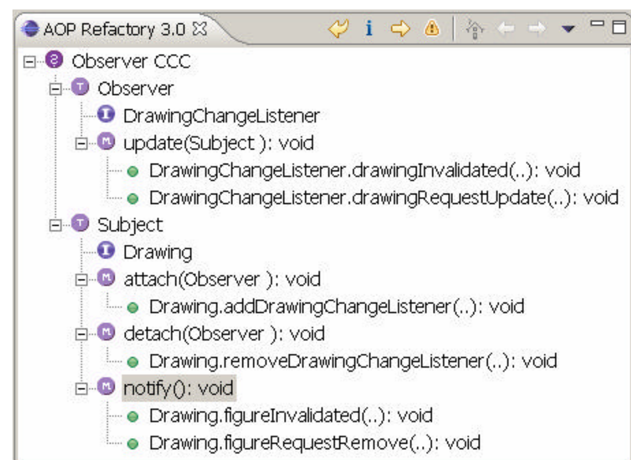

Figure 10: The Observer CCC with a sample mapping

[8] It is conceivable to add more information to the CCC model, such as the fact that *instance()* is a static method or that *Singleton*'s constructor is protected and search for types matching these criteria, so that the automapping can search the entire project for potential refactoring candidates.

In three of the four instances, we were aided in the mapping by the AutoMapper. One instance deviated so much from the pattern description that the CCC structure did not match the implementation. Lexical information in the abstract CCC description was helpful to distinguish between role methods attach and detach, for which the structural information is identical. Some of the lexical rules used to automap this CCC are shown below. The weights of each matching rule are summed to determine the score of the mapping candidate. We assigned the weights based on our expectations of the likelihood that the rule would be indicative:

Role method *attach()*:
- "add": weight=1
- "attach": weight=2
- "remove": weight=-1
- "detach": weight=-2
- "observer": weight=1
- "listener": weight=1

Role method *notify()*:
- "notify" weight=3
- "change" weight=1
- "observer" weight=1
- "listener" weight=1

Role type *Observer*:
- "observer" weight=2
- "listener" weight=2

In total, the three instances we refactored resulted in changes to 12, 10, and 23 types, respectively.

6. DISCUSSION

We demonstrated that our proof-of-concept tool can refactor some CCC in a non-trivial system. A number of issues remain to improve the applicability and robustness of the approach.

6.1 Variant Implementations of CCCs

The abstract description of the CCC is intended to capture the structural essence of several different ways to implement the CCC. However, some implementations of a CCC may still vary enough from the abstract definition so as not to be recognized as applicable by a tool implementing our approach. It is an open question as to how flexible the description of an abstract CCC should be to capture the different variants..

The JHotDraw variants of the Observer pattern are sufficiently diverse to discuss some of the range of implementation variants that need to be considered. Consider these five (selected) divergences from the pattern as documented in [6]:

1. Subject accepts only a single Observer
2. *notify()* role implementation appears only on concrete subjects but not on Subject interface
3. Multiple *notify()* and *update()* methods exist within the same pattern instance
4. the *observers* role field implementation is used outside of the pattern context
5. Non-observer content is stored in the concrete *observers* role field

It seems reasonable to expect the abstract CCC description to be flexible enough to capture at least variant one and two. The fifth variant, on the other hand, has so little in common with the original pattern, that a separate CCC description is necessary to capture the variant.

In the third variant, the subject provides multiple update methods for the same set of observers, which is a special case of the *push* model ([6], pp. 298). Although the structure of the pattern is similar to the CCC pattern description, the associated refactoring instructions did not cover this case. Since the structure is similar, the refactoring description should likely be extended to cover this variant.

In the fourth variant, references to the observers field from outside the pattern context can be detected by an impact analysis; before the field is considered for deletion, dealing with such references to it would constitute decision points and a conversation with the developer would be initiated.

Ideally, to keep refactoring descriptions simple, it seems better to address as much of the variant implementation within the refactoring engine as possible. Certain common themes in CCC variants can be covered this way, such as a missing explicit interface for certain design patterns. Other variants will have to be addressed by explicitly providing multiple CCC descriptions.

More experience with a broader base of CCC as they occur in different kinds of systems is needed to better extrapolate commonalities with respect to implementation alternatives and to handle those in the refactoring engine in a more general way.

6.2 Other CCC Refactorings

For this paper, we have investigated only one particular kind of CCCs, namely design patterns. Patterns make good candidate CCCs because many profit from an AOP implementation [10] and they have a clear (albeit flexible) structure to leverage. In particular, pattern elements are usually named and map cleanly to role elements. Role-based refactoring can conceivably be applied to restructure other CCC as well, as long as the CCC has at least some structure. A refactoring that replaces scattered calls to a particular method (e.g., a logging method) with an equivalent pointcut and advice is a simple example of a non-pattern CCC refactoring.

Similarly, the idea that complex refactorings can be planned via an abstract description of the concern (using roles) and a set of refactoring instructions is not restricted to OO-to-AOP refactorings. We plan to investigate “big refactorings” [5] as future work to get an idea of how useful the approach is for complex refactorings in general.

6.3 Suggesting Refactorings

Currently our approach requires the developer to choose an appropriate CCC refactoring. It is conceivable that the tool could be extended to integrate an aspect-mining approach (see for example [2, 26, 28], so that non-modularized CCC could be identified and refactorings could be suggested. Current aspect mining approaches do not use role abstractions, and picking an appropriate CCC refactoring would still be necessary. The refactoring tool’s library of CCC refactorings could direct mining approach, suggesting the CCCs for which to mine.

6.4 Describing and Executing Refactorings

Refactorings are described as elemental AOP refactoring steps, referring to the role elements identified in the CCC description. Currently, they are hard coded calls to a library of primitive refactorings we have developed. The declarative nature of these descriptions suggests that a language could be developed for them. This would allow us to perform some basic consistency checks during parsing, and would allow the tool to reason over the language, generating elements of the conversation structure and queries directly from the refactoring description.

6.5 Mapping Roles

Our current model for the auto-completion of mappings between roles and concrete program elements uses both structural and lexical information. This approach has made the mapping step tractable for the examples we have investigated. However, we may be able to improve this step by using semantic information about the internals of role elements, in particular role methods. For example, we know that within the scope of the *attach(Observer)* method in the Observer pattern, the argument object has to be added to some sort of data structure. In other words, a candidate method whose scope contains calls to an add(..), put(..), or insert(..) method of an aggregate data structure is a more likely candidate than one whose scope does not. This additional information could be used to rank candidate matching methods and to suggest likely matches to the developer.

We did consider using a subgraph-matching approach as utilized in [1], but this did not provide us with feedback on the level of single program elements, making it difficult to combine it with other analyses and utilizing the lexical information available.

6.6 Preservation of Behavior vs. Intent

As mentioned in section 2.4, some transformations may result in minor behavior variations to the code base. In such cases, preservation of intent takes the role of the (for traditional refactorings integral) preservation of behavior, in the sense that the transformation preserves the overall requirements of the system, but may change the semantics of the program locally.

As the impact analysis should point out any behavior variations to the developer, it is usually possible to avoid behavior implications by adjusting or relaxing the original CCC implementation before performing the refactoring. The tradeoffs are obvious: such initial refactorings require knowledge of the problem and additional effort, while otherwise the behavior of the system might vary slightly. Further analysis of cases that do not preserve behavior is future work

7. CONCLUSIONS

Transforming a scattered implementation of a crosscutting concern into a modular AOP implementation requires many changes to the code. In this paper, we have introduced an approach for refactoring CCCs based on roles. The roles allow the abstract description of the CCC; without this abstraction, each time a similar scattered implementation of a CCC appears in the code for a system, a refactoring would have to be built from scratch. We have shown that the approach is viable for non-trivial code by using our tool to refactor instances of three different design pattern CCCs in the JHotDraw graphical editing framework.

8. ACKNOWLDGEMENTS

This research was funded by NSERC and by IBM. We would like to thank Jonathan Sillito and the referees for their comments on drafts of this paper.

9. REFERENCES

[1] Baniassad, E. *Design Pattern Rationale Graphs: Linking Design to Source.* PhD Dissertation, University of British Columbia, Canada, 2002.

[2] van Deursen, A., Marin, M., and Moonen L. Aspect Mining and Refactoring. *First International Workshop on REFactoring: Achievements, Challenges, Effects (REFACE '03)*, at 10th Working Conference on Reverse Engineering (WCRE '03), Victoria, BC, Canada, 2003.

[3] Feature Exploration and Analysis Tool. Web site. www.cs.ubc.ca/labs/spl/projects/feat

[4] Ferrante, J., Ottenstein, K. J., and Warren, J. D. The Program Dependence Graph and its Uses in Optimization. In *ACM Transactions on Programming Languages and Systems*, 3(9): pp.319—349, July 1987.

[5] Fowler, M. Refactoring – Improving the Design of Existing Code. Addison-Wesley, Boston, MA, 2000.

[6] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[7] Griswold, W. G. *Program Restructuring as an Aid to Software Maintenance.* Ph.D. Thesis and Technical Report 91-08-04, University of Washington, WA, 1991.

[8] Hanenberg, S., Oberschulte, C., and Unland, R.. Refactoring of Aspect-Oriented Software. NetObject Days. Erfurt, Germany, 2003.

[9] Hanenberg, S., Unland, R. Roles and Aspects: Similarities, Differences, and Synergetic Potential, In: *Proceedings of the 8th International Conference on Object-Oriented Information Systems*, pp. 507 – 520, LNCS Volume 2425 / 2002. Springer-Verlag, London, UK, 2002.

[10] Hannemann, J. and Kiczales, G. Design Pattern Implementation in Java and AspectJ. In *Proceedings of the 17th Annual ACM conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '02)*, pp. 161-173. Seattle, WA, November 2002.

[11] Hannemann, J., Fritz, T., Murphy, G. Refactoring to Aspects – an Interactive Approach. Eclipse Technology eXchange (ETX) Workshop at OOPSLA 2003.

[12] Hannemann, J., Chitchyan, R., and Rashid, A. Report on the Workshop on Analysis of Aspect-Oriented Software. In: *Object-Oriented Technology: ECOOP 2003 Workshop Reader*, pp. 154 - 164. Buschmann, F., Buchmann, A. P., Cilia, M. A. (Editors). LNCS Volume 3013 / 2004. Springer-Verlag, Heidelberg, Germany, 2004.

[13] Hilsdale, E., Hugunin, J. Advice Weaving in AspectJ. In *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD '04)*, pages 26-35. Lancaster, UK, 2004.

[14] Iwamoto M. and Zhao, J. Refactoring Aspect-Oriented Programs, *4th AOSD Modeling With UML Workshop*, at UML '03, San Francisco, CA, October 2003.

[15] JHotDraw open source proje ct. Web site. www.jhotdraw.org

[16] Kendall, E. A. Role Model Designs and Implementations with Aspect-oriented Programming. In Proceedings of the ACM conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA ‘99), pages 353-369. Denver, CO, 1999.

[17] Kiczales, G., Lamping, J., Mendhekar, A., Maede, C., Lopes, C., Loingtier, J.-M., Irwin, J. Aspect-Oriented Programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP ’97)*, pages 220-242. Jyväskylä, Finland, 1997.

[18] Kiczales, G., Hilsdale, E., Hugunin, J, Kersten, M, Griswold, W. G., An Overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP ’01)*, LNCS Volume 2072 / 2001, pp. 327-353. Springer Verlag, 2001.

[19] Laddad, R. Aspect-Oriented Refactoring Series. Part 1 and 2. The Server Side, 2003. http://www.theserverside.com/

[20] Lieberherr, K., Lorenz, D.H., and Mezini, M. Programming with Aspectual Components. Technical report NU-CCS-99-01, College of Computer Science, Northeastern University, Boston, MA, 1999.

[21] Lieberherr, K., Lorenz, D.H., and Ovlinger, J. Aspectual Collaborations: Combining Modules and Aspects. In *The Computer Journal*, volume 46, issue 5: pp. 542-565, Oxford University Press, 2003.

[22] Masuhara, H. and Kiczales, G. Modeling Crosscutting in Aspect-Oriented Mechanisms. In *Proceedings of the 17th European Conference on Object-Oriented Programming (ECOOP ’03)*, pp. 2-28. Darmstadt, Germany, 2003.

[23] Monteiro, M. P., Fernandes, J. M. Object-to-Aspect Refactorings for Feature Extraction. Industry track paper at the 3rd International Conference on Aspect-Oriented Software Development. Lancaster, UK, 2004.

[24] Opdyke, W. F. Refactoring Object-Oriented Frameworks. PhD Dissertation, University of Illinois, IL, 1992.

[25] Robillard, M. P., and Murphy, G. C. Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies. In *Proceedings of the 24th International Conference on Software Engineering (ICSE ’02),* pp. 406-416. Orlando, FL, 2002.

[26] Shepherd, D., and Pollock, L. Ophir: A Framework for Automatic Mining and Refactoring of Aspects. Technical Report No. 2004-03. Dept. of Computer & Information Sciences, University of Delaware, Newark, DE, 2003.

[27] Tamai, T., Ubayashi, N. and Ichiyama, R. An Adaptive Object Model with Dynamic Role Binding. To appear in Proceedings of the *International Conference on Software Engineering (ICSE ’05)*, St. Louis, MO, 2005.

[28] Tonella, P., and Ceccato, M. Aspect Mining though the Formal Concept Analysis of Execution Traces. IRST Technical Report, May 2004.

[29] Wloka, Jan. Refactoring in the Presence of Aspects. *13th Workshop for PhD Students in Object-Oriented Systems (PhDOOS)*, at ECOOP ‘03, Darmstadt, Germany, 2003.

10. Appendix

To date, we have used the following aspect-oriented refactorings in the CCC refactorings we have investigated

- *Create Subaspect:* creates a concrete subaspect from an existing abstract aspect. We used this primitive when a reusable AOP pattern implementation in the form of a library aspect was available
- *Add Internal Interface:* most reusable AOP pattern implementations utilize empty interfaces internally to provide a primitive form of typing. This refactoring adds an aspect-defined interface to a concrete type in the system.
- *Replace Object Method with Aspect Method:* replaces a non-static method on a type with a static method on an aspect. This involves adding a parameter representing the target object of the original call.
- *Replace Method Call with Pointcut and Advice:* generates a pointcut and advice code that replaces explicit calls to a given method.
- *Replace Method with Intertype Method Declaration:* removes a method from a type and creates an appropriate inter-type declaration for it.
- *Replace Field with Intertype Field Declaration:* as above, but for fields.

A list of 24 aspect-oriented refactorings is proposed in [14], some of which are confirmed by our findings. A detailed comparison is not possible without a more complete description of the refactorings in [14]. Besides differences in their naming, we further confirmed *Add Internal Interface*, *Replace Method with Intertype Method Declaration,* and *Replace Field with Intertype Field Declaration*. The latter two are also mentioned in [23].