Generic Roles for Increased Reuseability

Andreas Mertgen

Institut für Softwaretechnik und Theoretische Informatik
Technische Universität Berlin
Ernst-Reuter-Platz 7
D-10587 Berlin
andreas.mertgen@tu-berlin.de

Abstract: Role-based programming, as in the *Object Teams/Java* (OT/J) language, aims to improve object-oriented programming with regard to separation of crosscutting or context-related concerns. Therefore, OT/J introduces class-like modules for roles and context, which connect common classes to build collaborations. However, since role and base objects are directly linked, it implies strong coupling and limited possibilities of reuse. This research aims to create a generic way of expressing connections between a collaboration and its base in order to further decouple modules and enhance their reusability. We introduce a quantification mechanism based on logic meta-programming in *Prolog* that allows using generic references to declaratively defined program elements, which are transformed to build valid OT/J code. We propose that the use of logic meta-variables improves the expressiveness and genericity of role-based programming.

1 Introduction

Separation of concerns is a key principle applied in designing complex, flexible, and reusable software systems. Today's programming languages need to provide features that strongly support modularization. In object-oriented programming (OOP), classes and objects are abstractions of real-world entities with state and behavior and form a well-accepted set of modules to implement functional concerns. However, there are certain dimensions of modularization where the concepts of classes and objects reach their limitations. Our research focuses on two such dimensions: (1) the modularization of crosscutting concerns and (2) modularization of collaboration and context. Both areas of concern cannot be clearly separated in a single class module. Crosscutting concerns are often non-functional concerns and are scattered around multiple classes or tangled together in a single module. Collaborations may involve several classes and instances and depend on a context; all collaboration members together build a strongly coupled composite. Several developments exist that address these concerns and improve modularization support in OOP.

Modularization of crosscutting concerns is addressed in aspect-oriented programming (AOP) [KIL+97]. An aspect is a module that defines *advices* and *pointcuts*. The *advice* construct

is generally similar to a method and defines additional or altered program behavior. *Pointcuts* specify where the advice behavior has to be applied in the influenced base program by describing a set of well-defined points in the program control flow - the so-called *join points*. A pointcut is defined by stating conditions about the structure, state and/or dynamic behavior of a program. The idea of influencing multiple points in a program with a single module is called *quantification*. Quantification is a key technique for modularizing crosscutting concerns [FF00]. Currently, the most prominent apect-oriented language is AspectJ [KHH+01].

Modularization of collaborations is addressed in role-oriented (RO) programming. A role describes the intersection of an entity with a context. Entities may exist outside a context, and their features are defined independently, as in the case of ordinary objects. A context builds a scenario, in which several entities assume specific roles in a collaboration. One example is an entity, `Person`, which plays a role `Employee` within a context, `Company`. Two relationships define a role (e.g., `Employee`): (1) a player relationship to a base (`Person`), which plays the role and (2) a containment relationship to a context (`Company`), which harbors the role. A role uses and extends its base. The context-specific state and behavior are attached to the context and its roles. Thus, all context and role-dependent concerns can be modularized independently and do not get tangled within the base modules. The notion of roles is used in the design and modeling of software, but it is hardly supported in mainstream programming languages. A language with explicit support for roles is Object Teams/Java [Her07], which is the focus of this paper.

In Object Teams, a context is represented by a so-called *team* class. Within a team, there may be several role classes, each of which may be bound to a base class via a *playedBy* relationship. For each base class instance, in each surrounding team, an instance of a bound role class may exist. The *playedBy* relationship between role and base class shows many similarities with the *extends* relationship between sub- and superclass. First, the role may delegate to all methods and fields of its base (so-called *callout* bindings). Second, the role may override any method of the base with specialized behavior (*callin* bindings). Third, variables declared with the role type are substitutes for variables declared with the base type (*translation polymorphism*).

2 Problem Description

Although role orientation in Object Teams shares several ideas with aspect orientation, few possibilities exist for quantification. So far, binding of roles to base classes, including callin bindings, have to be made explicitly. Although a programmer may select the bindings on the basis of specific criteria, which are based on program structure and optionally qualified by program state, since there is no abstraction for describing this selection, the programmer may not express the binding in a reuseable way. Because of this, a programmer encounters numerous problems, which are listed below, and this paper will seek to contribute to the solutions to these problems:

1. **Modularization and reuse.** The missing generalization feature of selection criteria implies that there is no use (nor reuse) of any selection logic for quantifying over binding elements. An abstract selection mechanism would allow quantification of such elements, thus enhancing modularization and reuse options.

2. **Evolution and fragility.** Explicit bindings imply a high coupling between role and base classes, which affects the robustness of the program. Thus, evolution of the base program requires a review of all affected roles. A declarative abstraction of these bindings would loosen the coupling and mitigate evolution issues.

3. **Maintenance.** Lack of quantification support is an inconvenience. Although it is possible to achieve the same results without using quantification, it entails coding overhead, which may be cumbersome and error-prone. Furthermore, a declarative definition of bindings can also contribute to the readability and documentation of the code.

The join point model for callin bindings in Object Teams is limited to *method call interception*, that is, the execution of a base method may be delegated to a corresponding role method, analogous to the *method overriding* between sub- and superclass. At first sight, from an aspect-oriented perspective, this seems to be a hindrance; however, on closer inspection, binding via method call interception fits in well with the ideas of roles and context. Callin bindings may be viewed as *contextual method overriding*. Keeping bindings at the granularity of methods prevents unintended interference with invariants, pre- and post-conditions, and encapsulation. The limitations of bindings reflect a trade-off between expressiveness, safety, and usability in language design.

By contrast, quantification in AOP, like AspectJ, utilizes an extensive join point model. Numerous pointcut designators exist to select join points by kind, scope, or context. In addition, the programmer may use wildcards to describe patterns; e.g., a pointcut `call(void set*(..))` would describe all calls of setter-methods in a single statement. However, the feasibility of such patterns simply relies on naming and structuring conventions. There is no relation with regards to content between the intention of the programmer and the semantics of a wildcard expression. For example, a call to a method named `setup` would also match the example pointcut, which may not have been intended by the programmer. This may lead to unintended mismatches and pose a problem during evolution, e.g., during refactoring, when a programmer renames or moves a method without considering the influence of such changes on the evaluation of pointcuts [KS04]. Thus, wildcards seem to be a "one-size-fits-all" approach, requiring the arranging of patterns to succeed in complex scenarios [GB03].

Nevertheless, the concept of quantification in AOP has proved to be very useful in addressing crosscutting concerns. Therefore, we believe that quantification can improve the power of an RO approach like Object Teams, even though the join point model is less sophisticated.

3 Motivating Example

To illustrate the motivation for this work, we examine a typical example of a non-functional, crosscutting concern: synchronizing UI views with their corresponding data model. The scenario involves objects (playing the role of observers), which have to keep track of the current state of other objects (playing the role of the observed subjects). Such tasks are commonly realized by using the observer pattern, where every subject is responsible for notifying its dependent observers about any relevant changes in its state. Triggering the notification is a crosscutting concern, influencing one or more mutator methods in each involved subject.

As a concrete example, consider a simple application that displays geometrical figures on the screen. If any change occurs in a figure, the displaying object must be notified and updated automatically. An object-oriented approach (see Figure 1) usually includes a (possibly abstract) superclass or interface (e.g., `Figure`), which is the root type for all classes of view-relevant objects (e.g., `Point` and `Line`). These classes represent the subjects and inherit a `Subject` class that serves two purposes: (1) maintaining subject-observer mappings (`add-` and `removeObserver`) and (2) triggering the update logic (`notify`). The latter is performed whenever a subject wants to report a state change, which leads to calling all its registered observers (`update`).

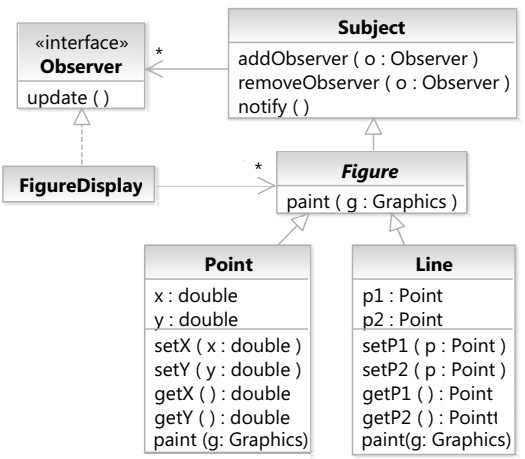

Figure 1: A simple system for displaying figures implementing the observer pattern

Variations of this example scenario are widely used in the field of aspect-orientation; the characteristics of the standard object-oriented pattern implementation in Java as well as the improvements of an aspect-oriented approach in AspectJ are demontrated in [HK02]. In terms of modularization of the crosscutting concern, the RO approach with Object Teams is quite similar to the aspect-oriented solution. The observer functionality is sourced out to an additional module, a *team* class, as shown in Listing 1.

The `FigureDisplay` class contains a list of figures to be displayed on the screen, reflecting their current state. Within this context, classes `Point` and `Line` play the role of subjects; for each, a role class is defined and bound to its base class *(lns. 8, 13)*. A base guard dynamically maintains the subject-observer mapping. An instance of `Point` should play the `PointSubject` role only if it is part of the team's figure list *(ln. 9)*; the same applies to `Line` instances *(ln. 14)*. The updating mechanism is realized with a callin binding to all mutator methods, the execution of one of these base class methods triggers the execution of an `update` in the team *(lns. 10, 15)*. No additional code is required. Any role-related code is defined within the module. Thus, `Figure` instances are completely unaware of their role as a subject and their original implementations are not required to change. The observer may also be implemented as a role. However, in this case, we prefer to implement the observer as a team because it is not only a role within a context; it actually **is** the context.

```
1  public team class FigureDisplay {
2      private List<Figure> figures;
3      ...
4      public void update() {
5          for(Figure fig : figures) {fig.paint(graphics);}
6      }
7
8      protected class PointSubject playedBy Point
9          base when (figures.contains(base)) {
10         update <- after setX, setY;
11     }
12
13     protected class LineSubject playedBy Line
14         base when (figures.contains(base)) {
15         update <- after setP1, setP2;
16     }
17 }
```

Listing 1: FigureDisplay

To avoid unnecessary updates (e.g., those caused by nested mutators), update calls should be buffered and accumulated, instead of being executed immediately. However, for the sake of simplicity, we refrained from optimizing and omitted all parts not relevant to the pattern mechanism.

As stated in [HK02], the structure of a pattern implementation contains common parts essential to all pattern instantiations and individual parts specific to each instantiation. The common parts of the observer pattern are as follows:

1. Roles for subjects and observers
2. Maintenance of subject-observer mapping
3. General updating mechanism (trigger)

The individual parts of the observer pattern are:

4. Binding of roles to certain objects
5. Binding of triggers to certain methods
6. Specific updating procedure of the observer

For the purpose of reuse in further developments, all common parts (1-3) may be extracted into an abstract class, whereas individual parts (4-6) need a concrete extension. This is good practice for the OO, AO, and RO approaches mentioned, but we argue that in this scenario (as in many others) there are additional opportunities for abstraction and reuse.

On closer examination of the observer `FigureDisplay`, we see that information about the individual parts' role and trigger bindings (4 and 5) is inherent to the updating procedure (6). Objects that may be considered for role playing are those accessed during `update` (members of the attribute `figures`); their relevant base classes are `Figure` and all its subclasses (4). Thus, a call to `Figure.paint` updates the display. Hence, all properties of a figure instance read within `paint` seem to be relevant for updating. Changes of exactly these properties should trigger the observer, which may be accomplished by binding all mutator methods of these properties (5).

Parts 4 and 5 pose a quantification issue. Although we find the required information inherent in the program structure, utilizing this information requires complex code analysis. A programmer could clearly describe the idea of the update-mechanism, e.g., *"trigger the update after the execution of any method that may cause a change of a field read somewhere in my update processing."* Such a statement could be reused in many similar cases, e.g., for updating a textual data representation in a table view; no further change is needed except for selecting the update process. However, existing language features are not capable of expressing the quantification logic in a programmer's mind. Neither code analysis nor the use of analysis results are supported by the languages presented so far in this paper.

4 Generic Modules

Our approach aims to enhance abstraction and reuse capabilities in scenarios similar to the one described above, where we prefer to reference program elements like classes and methods in a generic way to promote reuse in multiple scenarios. Therefore, we introduce logic meta-variables, which will be bound and replaced by the use of meta-programming. Our goal is to build an extended version of the OT/J language, called *Generic Object Teams* (GOT), which during a precompilation step transforms its code into valid OT/J code. This way, we ensure full compatibility with OT/J and prevent interference with existing tools and the compiler. As the transformation process implies, we rely on static program information and do not intend to use dynamic execution properties. First, the generic parts of GOT represent regular static elements and do not have any other effect on program execution. Second, dynamic dependencies may still be addressed by guard predicates, although they will only affect program execution and not the transformation. To handle meta-variables in GOT , we introduce three language constructs: (1) queries to declaratively describe sets of program elements; (2) matching statements to evaluate

queries and bind free variables; and (3) *per*-blocks to build the unit for applying meta-variables and code transformation.

4.1 Querying Program Elements

To facilitate the binding of meta-variables, we need to offer the option of querying various program elements about their properties and relationships, thus making the abstract syntax tree (AST) information available to the programmer. Several similar approaches that use XQuery, SQL or OCL [EMO04, SS08] exist, featuring a direct and readable syntax for single elements. However, they do not support transitive closures. Queries involving multiple elements may make frequent use of quantifiers (e.g., *forall* and *exists*) and include relations across tree branches, resulting in recursion and/or nested tree walking. Instead, we have chosen a logic-based representation in Prolog, which offers more concise and elegant expressions in such scenarios because information is globally available in facts without the need to navigate a tree structure. For each AST node type, there is a fact representation. For example, a method call is represented in a fact, *callT(#id, #enclosingMethod, #receiver, #method, [#args, ...])*, holding a unique id for the fact, a reference (by id) to the calling method, a reference to the expression representing the receiver of the call, a reference to the invoked method, and a list of references to argument expressions. Thus, a Prolog query may include meta-variables that get unified to every possible match in the factbase; e.g., the query *callT(_, Caller, _, Callee, _)* would match pairs of the meta-variables *Caller* and *Callee* to the method ids, such that the Caller invokes the Callee at least once in the program (using wildcards for irrelevant information). Our query module attempts to encapsulate logical queries in a manner as similar as possible to that in Java. A query is a special kind of method, marked with the *otquery* keyword (see listing 2) for processing meta-variables. A query is a conditional, built from facts or other queries.

```
1  otquery qrySubjects(?Method ?update, ?Class ?baseclass,
               ?Method ?setter) {
2      ?Field -readField;
3      ?Method -calledMethod, -executedMethod;
4      ?Method -executedMethod;
5      call(_, ?update, _, -calledMethod, _) &&
6      overrides(-executedMethod, -calledMethod) &&
7      readsField(-executedMethod, -readField) &&
8      setsField(?setter, -readField) &&
9      isMemberOfClass(?setter, ?baseclass)
10 }
```

Listing 2: A query for the declarative description of the observer's subjects

Two points about meta-variables need to be mentioned: (1) they are typed with a special set of types and (2) they are declared using a prefix. These points are discussed in detail as follows:

Typing of variables. Meta-variables may store values of certain types. To be consistent with typed languages OT/J and Java, the newly introduced meta-variables shall follow the rules of declaration. Since the set of possible types is different, they will be indicated by using a *"?"* prefix. Our type set includes *?Class*, *?Method* and *?Field* and others related to the AST. For more general purposes, the types *?String*, *?int* and *?boolean* are also included. Typing allows the generic parts of the code to be statically checked before the evaluation; e.g., in a playedBy-statement, we clearly expect a meta-variable of type *?Class*. Any other type could not possibly lead to a valid result.

Prefixes. Meta-variables get bound to a set of matches. Although a match-statement looks like a method, its parameters are either *in* or *out* as in Prolog; in contrast to Java's strict call-by-value method parameters. The prefixes allow narrowing the binding options of a variable: The - (*"minus"*) declares an *out*-parameter, meaning the variable must not be bound at evaluation time. In contrast, the + (*"plus"*) declares an *in*-parameter, which has to be bound to evaluate other dependent properties. The *?* indicates an *in*- or *out*-parameter.

4.2 Integration of Meta-variables and Transformation

To offer the aforementioned genericity for OT/J, meta-variables should be accepted in the least by the playedBy and callin-bindings, though they would generally be accepted in several other locations as well. Listing 3 shows a sketch of a generic role for the observer example. The base class of the `Subject` role is defined by the free variable `?b`, whereas the triggering callin is defined by the free variable `?m`. Both variables have to be declared and bound. An obvious location to do this is on the level of the enclosing team, assigning the team as the main unit for collaborations. Therefore, a statement *match*, which looks similar to a method, is appended to the team declaration. The match statement declares the variables as its parameters, enabling them to be used within the team body. Their bindings are defined in the match body by a query expression.

```
1  public team class FigureDisplay
2    match(?Class ?b, ?Method ?m){qrySubjects(update,?b,?m)} {
3    ...
4    per (?b) {
5      protected class -Subject playedBy ?b
6            base when (figures.contains(base)) {
7        per (?m) {
8              update <- after ?m;
9        }
10     }
11   }
12 }
```

Listing 3: Version of FigureDisplay with Generic Object Teams

The query for the observer example in listing 2 realizes the selection of program elements as intended by the programmer. In the match statement, the first parameter is bound to the `FigureDisplay.update` method, leaving the other parameters open as outputs, expecting a set of tuples for base-classes and setter methods. The call fact will match meta-variable `-calledMethod` to `Figure.paint`. Considering dynamic dispatch, we need all methods overriding the `Figure.paint` method. A query `overrides` (not included in the example) then binds the meta-variable `-executedMethod` to match `Point.paint` and `Line.paint`. Then, we need to identify the fields that are (indirectly) read in these methods, and the methods (and the classes they belong to), which mutate that fields. This way, we finally identify the relevant setter methods that trigger an update. The evaluated result of the match for the observer example will be a set of class/method tuples: $\{(Point, setX); (Point, setY); (Line, setP1); (Line, setP2)\}$.

The goal of the transformation is to change the team class according to the matches so that every code fragment dependent on a meta-variable is generated once for every match instance. In the class body, a meta-variable may only appear within a *per-block*. A per-block defines a non-empty set of meta-variables to be in its scope. Per-blocks may be nested; the inner block extends its set of meta-variables by the set of the outer block. A meta-variable may be put in place of a direct reference. In the example in listing 3, the playedBy relationship holds `?b` in place for the base class, and the callin-binding holds `?m` in place for base methods. Per-blocks control the transformation; for each tuple of matching values of a per-block's set of meta-variables, the per-block's body gets generated once in the transformation output with every occurrence of a meta-variable replaced by its matched value.

Every generated program element on the per-block level must use an unbound meta-variable for its name because simple names would collide if there is more than one instance in the generation output. These elements may be classes, methods, or fields. Such meta-variables are defined in place and are not declared in a matching statement. Their types are inferred and bound by the transformation, e.g., class `-Subject` is stated as a meta-variable. To prevent ambiguous expressions, a non-declared meta-variable cannot be referenced from outside the per-block; inside the per-block, it is unique and safe to use.

Finally, the code of the example will transform to a team with two role classes replacing `-Subject`, one played by `Point`, the other played by `Line`, both having two callin statements linking the update method to setter methods of their base class. The `FigureDisplay` class remains decoupled of the base classes and is robust with respect to evolution. For instance, consider adding a new property in one of the figure classes, e.g., a color, or adding a new figure class; in all cases, the team and the query will adapt the new situation without change. In addition, the template class may be used in other observer scenarios, e.g. managing the textual representation of the figures and their properties in a table. All that is required is to give different arguments for the match-query. Different match-instantiations of GOT team classes may be realized by inheritance, though the details of this mechanism are beyond the scope of this paper.

5 Related Work

A role-based approach explicitly targeting cross cutting concerns is presented in [HMK05]. The authors demonstrate the value of abstracting cross cutting concerns with roles. However, this approach focuses on refactoring design patterns using AOP. Selection mechanisms are based on lexical information and require manual adjustments. The approach does not propose a general-purpose role-based programming language and follows the already discussed pointcut principles. The fragility of pointcuts is a well-documented problem; approaches in this field aim to support the programmer by providing either mechanical assistance in maintenance of pointcut expressions [KGRX11] or feedback about the implicit effects of base code changes on pointcut evaluation [KS04]. Another aspect-oriented approach addressing the genericity of program elements is *LogicAJ* [RK04]. The research shows the limitations of wildcard matching and demonstrates the benefits of genericity by enhancing pointcuts and advices with meta-variables. LogicAJ was a major source of inspiration for the development of our current approach. Unlike Object Teams, LogicAJ is purely aspect-oriented and does not feature roles or context. Application of genericity is based on general AspectJ constructs, whereas Object Teams focuses on the modularization of roles and its special needs and opportunities.

There are several logic-based approaches in Prolog or Datalog for querying program structure [KHR07, HVdMdV05, JDV03]. In refactoring approaches, as in the *REFACOLA* language [SKvP11], logic programming is also used for program transformation. The approaches share the need to represent imperative program elements in a declarative way to enable them to efficiently query the database. The translation is done by describing AST nodes and relationships in terms of facts and rules. In refactoring, the goal is to make slight changes in program structure while retaining the observable behavior of the program. In contrast to refactoring, in Object Teams the input language differs from the output language and the transformation focuses on replacing and generating new, adapted program elements like classes and methods. The AO enhancement *tracematches* also makes use of free variables, but limits them to utilizing execution history, for which it introduces new pointcut idioms [AAC+05]. The *Alpha* language [OMB05] goes even further by providing Prolog queries over an extensive representation of the program, including dynamic properties of the program execution (making significant concessions regarding efficiency). Both approaches target a more sophisticated aspect-oriented join point selection, rather than modularization and reuse.

Meta-AspectJ (MAJ) is a structured meta-programming tool for generating AspectJ programs using code templates [ZHS04]. Partial artifacts of the code are evaluated to generate either AspectJ or plain Java code. In contrast to our approach, MAJ does not provide expressions to declaratively select program elements; instead references are gained by traversing the Java AST.

An approach to handling context-related concerns is *context-oriented programming* (COP) [HCN08]. In COP, *layers* build first-class entities that may modularize and control context-dependent behavior. Although layers support dynamic dispatching, activation and scoping, there is no intention to support quantification beneath the activation of layers. Layers build an explicit enhancement for classes and are intentionally scattered across modules.

6 Conclusions and Further Work

In this paper, we have discussed the potential for modularization and reuse beyond standard object- and aspect-oriented approaches, and we presented Generic Object Teams, an enhanced version of Object Teams, which is able to support these opportunities. GOT uses generative programming to replace manual search and adaption of program elements with automatic generation. Initially, writing GOT queries and teams requires extra effort, but in return, they provide enhanced reuse capabilities and are more robust towards evolution.

The implementation of a proof-of-concept prototype for GOT is work in progress. So far, we have extended the OT/J language with the required additional language features and build a representation (based on the Eclipse modeling framework (EMF)) of the GOT abstract syntax tree in Prolog facts. We are able to analyze the program structure using Prolog queries. Further work is required to create a more detailed design and implement the automated code transformation mechanism. The combination of different dimensions of reuse, namely inheritance and generics, pose several interesting challenges that will be addressed next. We believe that this combination of features possesses the potential for even more reuse than what we have shown so far.

References

[AAC+05] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding trace matching with free variables to AspectJ. *SIGPLAN Not.*, 40(10):345–364, 2005.

[EMO04] Michael Eichberg, Mira Mezini, and Klaus Ostermann. Pointcuts as Functional Queries. In *APLAS '04: Proceedings of the 2nd Asian Symposium on Programming Languages and Systems*, volume 3302 of *Lecture Notes in Computer Science*, pages 366–381. Springer, 2004.

[FF00] Robert E. Filman and Daniel P. Friedman. Aspect-Oriented Programming is Quantification and Obliviousness. In *Workshop on Advanced Separation of Concerns, OOPSLA*, 2000.

[GB03] Kris Gybels and Johan Brichau. Arranging language features for more robust pattern-based crosscuts. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 60–69, New York, NY, USA, 2003. ACM.

[HCN08] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented Programming. *Journal of Object Technology*, 7(3):125–151, 2008.

[Her07] Stephan Herrmann. A precise model for contextual roles: The programming language ObjectTeams/Java. *Appl. Ontol.*, 2(2):181–207, 2007.

[HK02] Jan Hannemann and Gregor Kiczales. Design pattern implementation in Java and aspectJ. *SIGPLAN Not.*, 37(11):161–173, 2002.

[HMK05] Jan Hannemann, Gail C. Murphy, and Gregor Kiczales. Role-based refactoring of crosscutting concerns. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 135–146, New York, NY, USA, 2005. ACM.

[HVdMdV05] Elnar Hajiyev, Mathieu Verbaere, Oege de Moor, and Kris de Volder. CodeQuest: querying source code with datalog. In *OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 102–103, New York, NY, USA, 2005. ACM.

[JDV03] Doug Janzen and Kris De Volder. Navigating and querying code without getting lost. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 178–187, New York, NY, USA, 2003. ACM.

[KGRX11] R. Khatchadourian, P. Greenwood, A. Rashid, and G. Xu. Pointcut Rejuvenation: Recovering Pointcut Expressions in Evolving Aspect-Oriented Software. *Software Engineering, IEEE Transactions on*, PP(99):1, 2011.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An Overview of AspectJ. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353, London, UK, 2001. Springer-Verlag.

[KHR07] Günter Kniesel, Jan Hannemann, and Tobias Rho. A comparison of logic-based infrastructures for concern detection and extraction. In *LATE '07: Proceedings of the 3rd workshop on Linking aspect technology and evolution*, page 6, New York, NY, USA, 2007. ACM.

[KIL+97] Gregor Kiczales, John Irwin, John Lamping, Jean-Marc Loingtier, Cristina Videira Lopes, Chris Maeda, and Anurag Mendhekar. Aspect-Oriented Programming. In *ECOOP '97: Proceedings of the 11th European Conference on Object-Oriented Programming*, pages 220–242. Springer-Verlag, 1997.

[KS04] Christian Koppen and Maximilian Stoerzer. Pcdiff: Attacking the fragile pointcut problem. In *First European Interactive Workshop on Aspects in Software (EIWAS)*, 2004.

[OMB05] Klaus Ostermann, Mira Mezini, and Christoph Bockisch. Expressive Pointcuts for Increased Modularity. In Andrew Black, editor, *ECOOP 2005 - Object-Oriented Programming*, volume 3586 of *Lecture Notes in Computer Science*, pages 214–240. Springer Berlin / Heidelberg, 2005.

[RK04] Tobias Rho and Günter Kniesel. Uniform Genericity for Aspect Languages. In *Technical Report IAI-TR-2004-4, Computer Science Department III, University of Bonn*. Dec 2004.

[SKvP11] Friedrich Steimann, Christian Kollee, and Jens von Pilgrim. A Refactoring Constraint Language and Its Application to Eiffel. In Mira Mezini, editor, *ECOOP 2011 - Object-Oriented Programming*, volume 6813 of *Lecture Notes in Computer Science*, pages 255–280. Springer Berlin / Heidelberg, 2011.

[SS08] Mirko Seifert and Roland Samlaus. Static Source Code Analysis using OCL. In Jordi Cabot and Pieter Van Gorp, editors, *OCL'08*, 2008.

[ZHS04] David Zook, Shan Shan Huang, and Yannis Smaragdakis. Generating AspectJ programs with Meta-AspectJ. In *Generative Programming and Component Engineering: Third International Conference, GPCE 2004, volume 3286 of LNCS*, pages 1–19. Springer, 2004.