Decoupling Context

Introducing Quantification in Object Teams

Andreas Mertgen

Berlin Institute of Technology

andreas.mertgen@tu-berlin.de

Abstract

In this paper, we propose role-oriented programming, which is realized in the language Object Teams/Java, as an alternative approach toward modularizing context-dependent concerns. We aim to integrate the benefits of quantification without introducing issues related to encapsulation and robustness. A language extension to Object Teams is presented by combining quantification with role-playing. It is achieved by querying the static program structure and transforming the code by using logic meta programming in Prolog. We discuss the query mechanism in detail in the text.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features; D.2.13 [*Software Engineering*]: Reusable Software

General Terms Languages, Design

Keywords Role-Oriented Programming, Object Teams, Logic Meta-Programming, Prolog

1. Introduction

Separation of concerns and code reuse are primary goals of the *object-oriented programming* (OOP) paradigm. *Inheritance*, including overriding and dynamic dispatching of methods, is one of its major concepts. It follows the Open-Closed principle [13], given that a module is open for extensions but closed for modifications of the interface. Inheritance is suitable for use in many applications for the decomposition of core concerns and it contributes to modularity by enabling well-structured hierarchies of objects. However, inheritance is too rigid to handle crosscutting concerns effectively. This is due to undesired code scattering and tangling caused by the need to dispatch to the same functionality in multiple locations.

Aspect-oriented programming (AOP), presented in [9], addresses this need by introducing a module *aspect*, in which *pointcuts* add a new mechanism for dispatching. Pointcuts declaratively describe a set of join points in a base program, where the aspect code applies. The base program may remain oblivious of the adaption and the required code can be incorporated in a separate module. The possibility to define a piece of code in a single module and to specify the conditions under which the code is going to be applied to other parts of the program is called *quantification*. This is a desirable feature when dealing with crosscutting concerns [1]. Although the obliviousness and the loose coupling of aspects contribute well to modularity, they are in conflict with the principles of encapsulation [16]. The ability of prominent AOP languages, such as AspectJ, to access and adapt the base code at (almost) any time and location without the restrictions of encapsulation increases the risk of violating pre-/postconditions or the invariants of modules, which remain secure until that point. Another potential problem, so called *pointcut fragility* [11], results from changes to the base code that can cause join points to incorrectly match or mismatch a pointcut. Pointcut fragility poses a threat to the evolution robustness of a program. Among the multiple reasons for this threat [2], one is the use of wildcards in pointcut expressions. Several wildcards rely on the lexical matching of identifiers, although their name is not inherently connected to the code they represent (except by coding conventions). For example, a pointcut expression, `set*(..)`, intended to match all setter methods, may inadvertently encompass a method named `setup`, as well. This may result in the giving up of obliviousness and arranging the base code in order to fit to the pointcut [3].

In this paper, we propose a different approach toward the Open-Closed principle, referred to as *contextual method overriding*, which is realized in *role-oriented programming*. We aim to enhance the role-oriented language Object Teams to support quantification without the degradation of encapsulation or robustness. We present a combination of quantification and role-playing in Object Teams, achieved by querying the static program structure and transforming the code by using logic meta-programming.

1.1 Role-playing in Object Teams

Role-oriented programming (ROP) aims to provide better support for mapping real-world entities to program module hierarchies. This is accomplished by emphasizing the fact that the same entity may play different roles according to the context, in which it is interacting. ROP allows the context-based dispatching of object behavior. Roles may improve the separation of concerns, particularly in complex programs, because context-related code can be encapsulated in separate modules, even though it affects the same objects. Roles enable objects to dynamically change interfaces and behavior.

An object-oriented language with an explicit support of roles is *Object Teams/Java* (OT/J), which is in the scope of this paper. Object Teams aims to support the collaboration of objects, and therefore, introduces two new types of class modules: *Roles* and *Teams*.

Roles feature two special relationships. In the first relationship, a *role* is played by a *base*. A role class defines another class to be its base (via *playedBy* binding). For example, for a role class, *Student*, we may choose a class, *Person*, as its base. Every runtime instance of the role class is associated with a corresponding instance of the base class. Base classes do not require any changes and are unaware of the adaption performed by a role. The relationship between the role and the base has many similarities with inheritance. First, the role may share properties of the base class (via *callout* bindings); second, the role may intercept method execution of the base and dispatch it to other code (via *callin* bindings, featuring *before*, *after* and *replace* the base method), and third a role is a legal substitute in every location, where an instance of the base is required (*translation polymorphism*). Callins can be viewed as the contextual overriding of methods, meaning that the dynamic dispatch depends on the current context, which is defined by roles and teams. Together, callins and callouts achieve true delegation between role and base objects. Role-playing relationships offer some flexibility, which is not offered by inheritance. A base object may dynamically add and remove role instances during runtime. The base object may also be associated with multiple independent role instances belonging to different role classes or even the same role class. Object Teams supports *gradual encapsulation*, which offers a flexible balance between encapsulation and decapsulation [5].

In the second special relationship, a role is harbored by a context, which defines the environment and the meaning of a role. For example, a *Student* exists only within the context of a *University*; otherwise, it is simply a *Person*. Contexts in Object Teams are reified as *Team* classes. Roles are inner classes of a team and therefore, every instance of a role is an inner instance of an instance of a team. Each team instance may be activated and deactivated at runtime, determining if the context-dependent behavior of the included roles should apply or not. Only roles of active team instances affect the behavior of base objects by callins. The effect of callins can be further restricted by *guard predicates*, which are boolean expressions declared using the keyword `when`.

2. Motivation

We motivate and demonstrate the proposed approach, which is discussed in the next section, with the example of an OT team class in listing 1. The team, `InitializationChecker`, provides the context for a role, `MonitoredCircle`, which adapts a base class, `Circle` (ln. 3). The role checks, whether a field *radius* of the base class was set before it was accessed for the first time. Therefore, the role defines callin bindings to intercept the execution of any setter-method that would modify the field (ln. 11) and any getter-method that would read its value (ln. 19). In the example, two methods are involved in each case, because the `Circle` class has a second property, i.e. *area*, which is derived from, and hence, dependent on, radius. The first callin binding (ln. 11) extends both setter methods of the base to be followed by the `setterCalled` method of the role. The second callin binding (ln. 19) extends both getter methods of the base to be preceded by the `checkInit` method of the role. After the execution of a setter-method, the role records the initialization of the field radius (ln. 8). This information is checked before the execution of any getter-method, throwing an exception in case the initialization of the field has not occurred in advance (ln. 15).

```
 1  public team class InitializationChecker {
 2
 3      protected class MonitoredCircle
           playedBy Circle {
 4
 5        private boolean isFieldSet = false;
 6
 7        protected void setterCalled() {
 8          isFieldSet = true;
 9        }
10
11        setterCalled <- after setRadius,
             setArea;
12
13        protected void checkInit() {
14          if (!isFieldSet) {
15            throw new Exception("Field
                 radius has not been
                 initialized");
16          }
17        }
18
19        checkInit <- before getRadius,
             getArea;
20      }
21  }
```

Listing 1. A policy enforcement context

The code might be applied as a part of policy enforcement in a dependency injection framework, where it is desirable to test whether the properties of an object have been initialized with reasonable defaults before accessing them.

In the current state, Object Teams requires explicit bindings in the playedBy, callout and callin relationships, as can be seen in the example in listing 1. The possibilities of quantification are limited to allowing an enumeration of methods to be bound to a callin method. There is no method to declaratively describe bindings as we find in pointcut expressions. The code should work fine in the specific example. However, consider the case in which we wish to monitor other classes (possibly involving multiple properties) and hence wish to reuse our solution. Thus far, parts of the `MonitoredCircle` class may be factored out to an abstract superclass, but only to a limited extent. This would not include the playedBy and callin bindings, because they require static bindings, which we want to keep flexible in this use case. Our new approach aims to provide possibilities for a declarative expression and the quantification of such role-playing relationships. To avoid interference with encapsulation, we do not intend to change the binding mechanism between the role and the base. In contrast to pointcut mechanisms, our approach aims to identify the program elements (classes, methods, fields) that can be used in such bindings, and enable a generic quantified definition of their values.

3. Approach

Our approach is implemented as a language extension called *Generic Object Teams* (GOT), which is realized as a plug-in in Eclipse 3.7. It was first presented in [12], which outlines the basic concepts of GOT. In this paper we present a different use case and discuss the core mechanisms of querying the code in detail.

GOT introduces meta-variables as legal representatives of program elements in role-playing bindings. Meta-variables are defined and bound by the use of logic meta-programming in *Prolog*. In a pre-compilation step, GOT code is transformed to standard OT code through the replacement of all meta-variables and the corresponding statements. In order to effectively control evaluation and transformation, GOT adds the following three language constructs to OT: (1) queries to identify program elements; (2) match statements to parameterize and evaluate queries; and (3) *per*-blocks to apply meta-variables in code and control the transformation.

In listing 2 we present a generic version of our previous example. The team class is extended with a *match* expression (ln. 2), which is similar to a query method. Meta variables are declared as parameters of the match expression. In contrast to normal parameters in Java, a meta-parameter may not only be *in* but also *out*, as in Prolog. This allows binding of a free variable by evaluation of the query statements in the method body. The in/out-character is indicated by a prefix of the identifier: the prefix + indicates *in*, the prefix - indicates *out*, and the prefix *?* allows both *in* or *out*. The notation follows the standard Prolog notation. Moreover, meta-variables in GOT are typed with a special set of types to enable static type checking. The type names also use the prefix, *?*, to better distinguish them from normal types. The meta-variables are bound by query expressions in the match body (ln. 4); queries are discussed in more detail in the next section.

```
1  public team class GenericChecker
2    match(?Class ?base, ?Field ?f, ?Method
         ?getter, ?Method ?setter)
3    {
4      findProperties(?base, ?f, ?getter,
          ?setter)
5    }
6  {
7    per (?base) {
8      protected class -Monitored playedBy
          ?base {
9        per (?f) {
10         private boolean -isFieldSet =
              false;
11         protected void -setterCalled() {
12           -isFieldSet = true;
13         }
14         per (?setter) {
15           -setterCalled <- after ?setter;
16         }
17         protected void -checkInit() {
18           if (!-isFieldSet) {
19             throw new Exception("Field " +
                  ?f.getName() + " has not
                 been initialized");
20           }
21         }
22         per (?getter) {
23           -checkInit <- before ?getter;
24         }
25       }
26     }
27   }
28 }
```

Listing 2. Generic version of InitializationChecker

After evaluation, the match statement must deliver a set of tuples of matching values for the four meta-variables. For example, applying it to the `Circle` class, we want to identify two tuples: {*(Circle, radius, getRadius, setRadius); (Circle, radius, getArea, setArea)*}.

Within the team class, *per*-blocks build the scope for applying meta-variables. A per-block declares a non-empty set of meta variables. The inner parts of a per-block can be interpreted as a template. During transformation, for each matching tuple of values of these meta-variables, one instance of the template code within the per-block is copied into the final output code, with every occurrence of a meta-variable replaced by the corresponding value. A nested *per*-block depends on its enclosing block, and hence, may generate multiple instances for one instance of the enclosing block. For example, in listing 2, the role class, `-Monitored`, is generated once for each class that was matched to the meta-variable, `?base` (ln. 7 et seq.). Within the role, for each field of the base class matched to `?f`, a boolean field, `-isFieldSet`, and a pair of methods, `-setterCalled` and `-checkInit`, appear (ln. 9 et seq.). Furthermore, any getter- and setter-method related to field `?f`, captured in `?getter` and `?setter`, is bound to callins (lns. 15 and 23). A program element introduced by per-blocks, such as role class -Monitored, must have an unbound meta-variable as its name, which does not need declaration. It cannot be referenced from outside the block; every instance of the program element generated during transformation receives a name that is unique in the scope of the surrounding element.

Although our focus is the use of meta-variables in role-playing relationships, parts of our use case might need additional information, which applies at other locations of our code, e.g. when we want to access meta-information, like the name of a matched field as in the example (line 19 of listing 2). In this case, there is a need for a trade-off. On the one hand, we want to support the programmer in his task; on the other hand, we want to retain encapsulation. We decided to grant access to meta-variables outside of specifically defined locations (e.g., playedBy) by using reflection, making the decision explicit to the programmer. A meta-variable representing a program element is interpreted as a variable declared with a corresponding type of `java.lang.reflect` package. For example, the `?f` meta-variable can be used like any variable of `java.lang.reflect.Field`, and therefore, grants a method, `getName`, to access the name of the element (ln. 19). The transformation replaces the meta-variable with a normal variable of the appropriate type and includes a method, which uses the reflect API to initialize the variable, in the output code. The result of the example in listing 2 after transformation with the aforementioned query match looks like the code in listing 1, except for the inclusion and call of an additional method for the reflection based access to the radius field.

The advantage of the GOT approach is that it is completely decoupled from any base class, except for the matching query. Furthermore, the team class is now applicable to host multiple roles for different base classes with multiple fields without change. The resulting code in listing 1 would change in two ways: first, there would be multiple role classes in the team, one for each matching base class; second, within a role, there would be multiple sets of fields and methods dedicated to an initialization check, one for each matching field of the enclosing base class. We are able to control application of a generic team by simply adjusting the query. The initial effort required for coding genericity should reduce coding efforts with regard to quantification and reuse in different scenarios.

3.1 Queries

The purpose of queries is to enable programmers to accurately define program elements that are used in a role-playing relationship. We decided to use the logic programming language *Prolog*, to implement queries. Logic programming offers a very concise and elegant way of expressing queries, including multiple free variables and transitive closure. Furthermore, it emphasizes expressions addressing the structure of program elements and helps avoiding pitfalls of lexical matching that may increase fragility.

To provide maximum (static) information to the programmer, the entire *abstract syntax tree* (AST) of the program is transformed into Prolog facts. For example, a class is represented by a fact, *classT(#id, 'name', [#definitions], ...)*, which holds a unique ID for the fact, the name of the class, a list of IDs referencing other facts representing the definitions of that class, and some more information.

To integrate queries in GOT, we have introduced a new kind of method marked with the keyword *otquery* instead of a return type. These methods can be collected in query classes, so programmers may build and use query libraries.

One option to construct a query is by directly quoting a Prolog statement in an annotation, as shown in listing 3. The query `classOfName` finds classes and their names by matching *classT* facts. The query only provides the interface; the code itself is pure Prolog. Such queries allow access to the complete factbase at the level of the Prolog language.

```
@Prolog("classOfName(ID, Name) :- classT(
    ID, Name, _, _, _, _, _, _, _, _, _)")
public otquery classOfName(?Class ?c, ?
    String ?name) {}
```

Listing 3. A query to match a class to a name

If a core of queries is already existent, a more preferable option to construct a query is by combining other queries, because expressions are statically type checked. An example is given in listing 4. The query, `findProperties`, is aligned to match tuples of fields and its corresponding getter- and setter-methods in a class, as required in the `GenericChecker`.

```
otquery findProperties(?Class ?c, ?Field
    ?f, ?Method ?getter, ?Method ?setter)
{
    readsField(?getter, ?f) &&
    setsField(?setter, ?f) &&
    isMemberOfClass(?setter, ?c) &&
    isMemberOfClass(?getter, ?c) &&
    isMemberOfClass(?f, ?c)
}
```

Listing 4. A query to match getter- and setter-methods related to a property

During the transformation process, a query is translated to and evaluated in Prolog. The GOT transformer bridges the gap between the two worlds. Queries are extended to ensure the type conformity of matched values. Furthermore, GOT program elements are mapped to Prolog IDs and vice versa. For example, the programmer may pass an argument `Circle.class` to a parameter typed with `?Class`.

4. Related Work

Several logic-based approaches that provide querying engines based on Prolog or similar exist [4, 7, 10, 14]. The approaches share the need to declaratively define and match program elements. A common goal is to offer a more sophisticated join point selection. Another aspect-oriented approach that addresses genericity by the introduction of meta-variables is *LogicAJ* [15]. The research aims to overcome the limitations of wildcard matching and was a major source of inspiration for the development of our current approach. However, all these approaches are based on classic AOP and do not feature roles or context; breaking encapsulation is still an issue.

An approach to address context-related concerns is *context-oriented programming* (COP) [6, 17]. COP approaches introduce first-class enhancements for classes that offer context-dependent method dispatch. Although these approaches support dynamic dispatching and activation, they do not involve genericity of program elements.

A structured meta-programming tool for generating AspectJ programs using code templates is *Meta-AspectJ* (MAJ) [18]. In contrast to our approach, MAJ does not provide expressions to declaratively define program elements.

The fragility of pointcuts is a well-documented problem [2]. Several tools were presented, which analyze pointcut evaluation and provide feedback to the programmer to prevent mismatches [8, 11].

5. Conclusions and Future Work

In this paper, we have discussed the problems of AOP approaches with regard to encapsulation and robustness. We have proposed role-oriented programming in Object Teams as an alternative solution toward context-dependent concerns and introduced the language extension Generic Object Teams, which adds logic meta-variables and improves decoupling and quantification.

Several areas of our design remain to be explored in further detail. Specifically, the co-existence of inheritance with the newly introduced genericity poses a challenge, because both target modularization and reuse, and hence, need cooperation without interference. We plan to address these issues in a more sophisticated version of our transformation model.

References

[1] R. E. Filman and D. P. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Workshop on Advanced Separation of Concerns, OOPSLA*, 2000.

[2] P. Greenwood, A. Rashid, and R. T. Khatchadourian. Contributing factors to pointcut fragility. In *3rd Workshop on Assessment of Contemporary Modularization Techniques (ACoM.09)*, 2009.

[3] K. Gybels and J. 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.

[4] E. Hajiyev, M. Verbaere, O. de Moor, and K. 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.

[5] S. Herrmann. Gradual encapsulation. *Journal of Object Technology*, 7(9):47–68, 2008.

[6] R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3):125–151, 2008.

[7] D. Janzen and K. 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.

[8] 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.

[9] G. Kiczales, J. Irwin, J. Lamping, J.-M. Loingtier, C. V. Lopes, C. Maeda, and A. Mendhekar. Aspect-oriented programming. In *ECOOP '97: Proceedings of the 11th European Conference on Object-Oriented Programming*, pages 220–242. Springer-Verlag, 1997.

[10] G. Kniesel, J. Hannemann, and T. 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.

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

[12] A. Mertgen. Generic roles for increased reuseability. In S. Jähnichen, A. Küpper, and S. Albayrak, editors, *Software Engineering*, volume 198 of *LNI*, pages 131–142. GI, 2012.

[13] B. Meyer. *Object-Oriented Software Construction - 2nd ed.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2st edition, 1997.

[14] K. Ostermann, M. Mezini, and C. Bockisch. Expressive pointcuts for increased modularity. In *ECOOP 2005*, volume 3586 of *Lecture Notes in Computer Science*, pages 214–240. Springer Berlin / Heidelberg, 2005.

[15] T. Rho and G. Kniesel. Uniform genericity for aspect languages. In *Technical Report IAI-TR-2004-4, Computer Science Department III, University of Bonn*. Dec 2004.

[16] F. Steimann. The paradoxical success of aspect-oriented programming. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 481–497, New York, NY, USA, 2006. ACM.

[17] J. Vallejos, S. González, P. Costanza, W. De Meuter, T. D'Hondt, and K. Mens. Predicated generic functions: Enabling context-dependent method dispatch. In E. W. Benit Baudry, editor, *Proceedings of the 9th international conference on Software Composition*, number 6144 in Lecture Notes in Computer Science, pages 66–81. Springer Verlag, 2010.

[18] D. Zook, S. S. Huang, and Y. 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.