A Case for Statically Executable Advice: Checking the Law of Demeter with AspectJ*

Karl Lieberherr David H. Lorenz Pengcheng Wu

College of Computer & Information Science
Northeastern University
360 Huntington Avenue 161 CN
Boston, Massachusetts 02115 USA
{lieber,lorenz,wupc}@ccs.neu.edu

ABSTRACT

We define a generic join point model for checking the Law of Demeter (LoD). Join points are trees, pointcuts are predicates over join points, and advice is checked statically similar to how **`declare warning`** is checked in AspectJ. We illustrate how the joint point form is mapped to the object and class forms of LoD, and provide an implementation in AspectJ that approximates LoD's class form by dynamically checking a particular execution using only the join points' static part. The paper proposes two ways to extend AspectJ to provide access to lexical join points directly. The first proposes statically executable advice and pointcut designators over lexical join points. The second proposes statically executable meta-advice over the exposed abstract syntax tree of the program and using Demeter style traversals to mirror AspectJ pointcuts.

1. INTRODUCTION

The Law of Demeter (LoD) [9] is a style rule that improves the quality of object-oriented (OO) code [10]. Call sites and message sends in OO programs constitutes coupling between classes. LoD states which couplings are acceptable and which are best avoided. Informally, LoD states that an object should only talk to "closely related" objects, thus leading to less coupled OO systems [7].

There are three good reasons why one would want to check LoD using aspect-oriented programming (AOP):

1. Detecting violations is a cross-module concern [17].
2. LoD is easy to express in a join point model.
3. Detecting LoD violations is an interesting, non-trivial application of aspect technology helping to drive it further.

*This work was supported in part by the National Science Foundation (NSF) under Grants No. CCR-0098643 and CCR-0204432, by DARPA and BBN under agreement F33615-00-C-1694, and by an Eclipse Grant from OTI.

AOSD 2003 Boston, MA USA
© ACM 2003 1-58113-660-9/03/002...$5.00

In this paper we define LoD in terms of predicates over join points, and we use the general purpose programming language AspectJ [11, 8] to check for violations. It is natural to dynamically check LoD with AspectJ. Checking LoD's dynamic forms is useful but the set of violations detected depends on the input. We identify *statically executable advice* as a small improvement that could make AspectJ capable of checking also LoD's static forms.

In a nutshell, we wish we were able to express LoD as an abstract pointcut designator named `LoD`, which is made concrete in a subaspect, and declare `!LoD` to be a violation:

Listing 1: LoD violation aspect

```
abstract aspect Violation {
  abstract pointcut LoD;
  declare warning: !LoD: "LoD Violation";
}
```

Not surprisingly, this attempt is futile in the current implementation of AspectJ. [1] The logic required to detect LoD violations cannot be associated with the pointcut declared by the `Violation` aspect in Listing 1, not even for LoD forms which are statically checkable. AspectJ lets you only check statically whether or not a statically determinable pointcut designator is the empty set.

What is missing in AspectJ to make the `Violation` aspect work is a more expressive statically determinable pointcut designator and statically executable advice language that supports arbitrarily complex logic at compile time as long as only statically available information is accessed.

To show that AspectJ's join point model is expressive enough to describe lexical forms of LoD, we present an implementation of a statically checkable form of LoD in AspectJ, which uses dynamic checking but with the following restrictions:

- The advice is defined only on statically determinable pointcuts.
- The advice accesses only the static part of the join points.

Although we use advice for this implementation, we argue that it can be transformed to *statically executable advice*, which can be applied at compile time.

[1] We are referring to version 1.0.6 of AspectJ.

Outline. The rest of the paper is organized as follows. In Section 2 we describe background on LoD. In Section 3 we formulate LoD using join point trees. Section 4 introduces an AspectJ program for checking LoD's object form, and Section 5 gives a dynamic version of the checker for the class form. In section 6, we propose two extensions to AspectJ. Section 7 concludes.

2. BACKGROUND

LoD has had many different formulations, but the common denominator is that in order to reduce dependencies between objects (or classes), an object should only send messages to a certain set of "closely related" objects; the motto is "don't talk to strangers."

Checking LoD violations is a cross-cutting concern: it involves checking all method calls in a program. LoD has a class form (CF) and an object form (OF). CF states that the code of a class's method must rely only on the class's other methods or methods of the classes of its arguments, instance variables (data members), classes used to locally instantiate instances, and the classes that are return types of methods in the class. Validating CF can be done statically by parsing the source code.

OF states that an object can only send messages to itself, its arguments, its instance variables, a locally constructed object, or a returned object from a message sent to itself. OF is intended to be more restrictive than CF in the sense that OF cares about particular objects while CF only cares about types. Therefore, an OF violation is not necessarily a CF violation. However, a CF violation is not necessarily an OF violation either (e.g., typecasting might violate CF without violating OF.) Even when the static type of an object is a static LoD violation, the dynamic type might not be a dynamic LoD violation.

Other researchers have written LoD checkers. Naftaly Minsky and his team, working on law-governed systems, used their Darwin-E environment to check different relaxed versions of CF [14]. We illustrate that behavior similar to what can be expressed in Darwin-E for building law-governed systems can be achieved by using AspectJ. This both creates a set of new applications for AspectJ and it makes the work on law-governed systems more accessible through a general purpose language.

Basili *et al.* analyze several object-oriented design metrics as quality indicators of a program [3]. One of the design metrics they analyze is *coupling between object classes* (CBO). A class *A* is said to be coupled to another class *B* if *A* uses *B*'s member functions and/or instance variables. The CBO of a class is the number of classes to which the class is coupled. One of their hypotheses is that highly coupled classes are more fault-prone than weakly coupled classes and they find through statistical analysis of many programs that CBO-related hypothesis is significant. LoD, as a style rule, limits CBO. Therefore we conclude that programs that violate LoD are more fault-prone than programs that follow LoD.

ArchJava [1] enforces architectures on Java programs detects communication integrity violations. The architecture language of ArchJava defines components and connections and the implementation must conform to the architecture. Aldrich *et al.* postulate [1, Hypothesis 5]: "It will be relatively easy to use ArchJava to express the software architecture of an object-oriented program whose source code obeys the LoD." We believe that AspectJ, with the additions proposed in this paper, could become an excellent language to enforce software architectures.

In another related work, Deters [4] presents a solution to uses AspectJ to harvest runtime information for subsequent off-line analysis. The aspects that collect information are concerned with references between objects, constructor calls and field references and therefore use similar pointcuts as our LoD checkers.

3. JOIN POINT PREDICATES

In this paper we show that LoD can be expressed as a predicate on join point trees. There are two kinds of join points. A lexical join point is a site in the program text. A dynamic join point is an event in the execution of the program. *Shadow* is a function from dynamic to lexical join points, which maps a dynamic join point to its lexical shadow [13].

3.1 Predicates over join points

Let $\mathcal{L}_\mathcal{P}$ denote the set of lexical join points in a program $\mathcal{P}$. Let $\mathcal{D}_\mathcal{E}$ denote the set of dynamic join points in an execution $\mathcal{E}$ of $\mathcal{P}$. A dynamic join point contains both static and dynamic information. For a dynamic join point $d \in \mathcal{D}_\mathcal{E}$, we define $\sigma_\mathcal{P}(d)$ to be the static part of d (which is also the shadow of d.)

We define an equivalence relation over the dynamic join points in $\mathcal{D}_\mathcal{E}$ as $d_1 \sim d_2$ iff $\sigma_\mathcal{P}(d_1) = \sigma_\mathcal{P}(d_2)$, and denote by $[d]$ the equivalence class of d. We shall assume $\mathcal{L}_\mathcal{P}$ to be [2] the set of equivalence classes of $\sim$. That is, each lexical join point is considered to be the set of dynamic join points it shadows, and the equivalence class of a dynamic join point d is considered to be the lexical join point $[d]$.

Given a predicate $\pi_\mathcal{L} : \mathcal{L}_\mathcal{P} \rightarrow bool$ on lexical join points in $\mathcal{P}$, we can construct a predicate $\pi_\mathcal{D} : \mathcal{D}_\mathcal{E} \rightarrow bool$ on dynamic join points in an execution:

$$\pi_\mathcal{D}(d) = \pi_\mathcal{L}([d]) \tag{1}$$

An execution $\mathcal{E}$ of a program $\mathcal{P}$ satisfies the dynamic predicate $\pi_\mathcal{D}$, if all dynamic join points in $\mathcal{E}$ satisfy $\pi_\mathcal{D}$:

$$\tau_\mathcal{D}(\mathcal{E}) = \bigwedge_{d \in \mathcal{E}} \pi_\mathcal{D}(d) \tag{2}$$

A program $\mathcal{P}$ satisfies the dynamic predicate $\pi_\mathcal{D}$, if all executions of $\mathcal{P}$ satisfy $\tau_\mathcal{D}$:

$$\tau_\mathcal{L}(\mathcal{P}) = \bigwedge_{\mathcal{E}(\mathcal{P})} \tau_\mathcal{D}(\mathcal{E}) \tag{3}$$

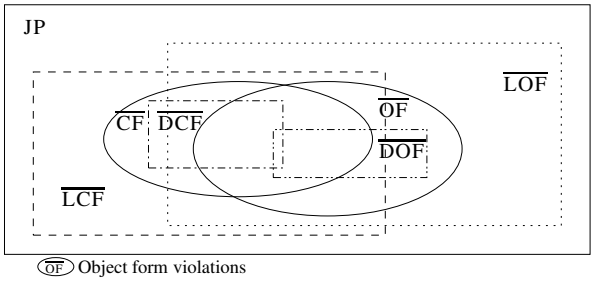
Given a dynamically checkable predicate $\tau_\mathcal{D}$, it is desirable to have $\tau_\mathcal{L}$ statically checkable. We can approximate $\tau_\mathcal{L}$ in AspectJ for a dynamic predicate $\pi_\mathcal{D}(d) = \pi_\mathcal{D}(\sigma_\mathcal{P}(d))$ that only accesses the static part of the join point. The paper makes the observation that advice defined over the equivalence classes of statically determinable join points can be statically executed at compile time.

For a program $\mathcal{P}$ and input $\mathcal{I}$ we construct the dynamic join points of the execution of $\mathcal{P}$ on $\mathcal{I}$ and the lexical join points produced from the shadow of the dynamic ones. We derive from LoD: LCF (Lexical Class Form), DCF (Dynamic Class Form), CF (Class Form), DOF (Dynamic Object Form), LOF (Lexical Object Form) and OF (Object Form). Let $\pi_\mathcal{D}$ be *class form predicate of LoD*, then by Eq. (2) we derive $\tau_\mathcal{D}$ to be DCF and by Eq (3) we derive $\tau_\mathcal{L}$ to be CF. Similarly, let $\pi_\mathcal{D}$ be *object form predicate of LoD*, then by Eq. (2) we derive $\tau_\mathcal{D}$ to be DOF and by Eq. (3) we derive $\tau_\mathcal{L}$ to be

[2]For simplicity, we ignore lexical join points that don't shadow any dynamic join point.

OF. LCF and LOF are respectively class and object form LoD predicates, applying static checking. The difference between DOF (DCF) and OF (CF) is that DOF (DCF) only guarantees that the program $\mathcal{P}$ doesn't violate the object (class) form in the execution on a given input $\mathcal{I}$. One difference between LOF (LCF) and OF (CF) is that LOF (LCF) also checks lexical join points in dead code that might not be reached by any dynamic join point.

Figure 1 shows the relationships between the violation reports by different forms of LoD predicates for a program $\mathcal{P}$ and input $\mathcal{I}$.



(OF) Object form violations
(CF) Class form violations
LCF: Violations reported by static checking the class form
LOF: Violations reported by static checking the object form
DCF: Violations reported by dynamic checking the class form
DOF: Violations reported by dynamic checking the object form
JP: All call sites

Figure 1: The relationship between different LoD predicates.

3.2 Law of Demeter over Join Points

The key abstract data type to formulate a generalized LoD checker is an abstract join point (tree). An abstract join point consists of a target, a list of args, a result, and a list of abstract join points.

```
JP ::= [target:α] args:{α}* [result:α] children:{JP}*;
```

We define a Join Point Form (JPF) of LoD:

DEFINITION 1 (JPF). *The* Join Point Form *of LoD requires that for each join point J, the target of J must be a potential preferred supplier to J.*

DEFINITION 2 (POTENTIAL PREFERRED SUPPLIER). *The set of* potential preferred suppliers *to a join point J, child of the enclosing the join point E, is the union of the following sets:*

- *Argument Rule: the argument list of the enclosing join point E, parent to J;*
- *Associated Rule: the results of the siblings of J which do not have a target or whose target is the target of the enclosing join point E, parent to J.*

More formally, the signature of an abstract join point is:

```
datatype α jp = jp of α * α list * α * α jp list;
```

This signature is parameterized by the type α, where α can stand for an object (for modeling a dynamic join point) or a class (for modeling a lexical join point) or any other type (for other unexplored forms of LoD).

The basic constructor of the α jp abstract data type is:

```
jp: α * α list * α * α jp list → α jp
```

with accessors:

```
getTarget: α jp → α
getArgs: α jp → α list
getResult: α jp → α
getChildren: α jp → α jp list
```

and predicates:

```
contains: 'α list * 'α → bool
isChildOf: 'α jp * 'α jp → bool
hasSelfishChild: 'α jp * 'α → bool
```

The algebraic specification is given in Listing 2, accounting also for the possibility that the optional target and result are missing.

Using an AspectJ-like syntax and the abstract data type, JPF can be expressed as a pointcut over a traversal of the abstract join point tree [5]. In Listing 3, **thisJoinPoint** denotes the current join point, and **thisEnclosingJoinPoint** denotes its parent, such that `isChildOf(thisJoinPoint,thisEnclosingJoinPoint)` is always true. The OO syntax $target.f(arg_1, \ldots)$ is used instead of $f(target, arg_1, \ldots)$.

Listing 3: LoD aspect

```
aspect LoD extends Violation {
  pointcut LoD(): //LoD definition
    ArgumentRule()
    || AssociatedRule();
  pointcut ArgumentRule():
    if(thisEnclosingJoinPoint.getArgs()
      .contains(thisJoinPoint.getTarget()));
  pointcut AssociatedRule():
    if(thisEnclosingJoinPoint
       .hasSelfishChild(thisJoinPoint.getTarget()));
}
```

The pseudo aspect in Listing 3 formulates LoD as a predicate on join point trees consisting of only one parameterized kind of join point (which we instantiate later to become an object-based or a class-based join point). The LoD aspect extends the Violation aspect (Listing 1) by giving the definition of the LoD pointcut in terms of *Argument Rule* and *Associated Rule*, which in turn are implemented as pointcuts. The **declare warning** defined in Violation provides a notification of a violation if `!LoD()` is non-empty.

LoD is a "pseudo" aspect because it cannot run in the current implementation of AspectJ, which doesn't allow **declare warning** to be defined on a pointcut with an **if** expression. The pointcut ArgumentRule and AssociatedRule select the "good" join points in the entire join point tree. ArgumentRule selects those join points whose target is one of the arguments of the enclosing join point; AssociatedRule selects those join points whose target is in the set of locally returned α's, the result α's of the method call on the enclosing join point target, and the α's created in the enclosing method body. [3] A concrete implementation of AssociatedRule should account for the direct part α's of the enclosing join point target, and the presence of aliasing.

Definition 1 (JPF) is consistent with OF for a multi-dispatch (actually, predicate-dispatch) language called Fred [15]. The JPF formu-

[3]There is no target for constructor calls.

Listing 2: Algebraic specification of the join point abstract data type

```
datatype 'a optional = empty | data of 'a;
datatype 'a jp = jp of 'a optional * 'a list * 'a optional * 'a jp list
fun getTarget (thisJoinPoint as jp(target,_,_,_)) = target;
fun getArgs (thisJoinPoint as jp(_,args,_,_)) = args;
fun getResult (thisJoinPoint as jp(_,_,result,_)) = result;
fun getChildren (thisJoinPoint as jp(_,_,_,children)) = children;
fun contains([],_) = false
  | contains(head::tail,element)= head=element orelse contains(tail,element);
fun isChildOf (thisJoinPoint,(parent as jp(_,_,_,[]))) = false
  | isChildOf (thisJoinPoint,(parent as jp(target,args,result,child::children))) =
      thisJoinPoint=child
      orelse isChildOf(thisJoinPoint,jp(target,args,result,children));
fun hasSelfishChild ((parent as jp(_,_,_,[])),_) = false
  | hasSelfishChild ((parent as jp(target,args,result,
                      (child as jp(childtarget,_,childresult,_))::children)),receiver) =
      receiver=childresult andalso (target=childtarget orelse target=empty)
      orelse hasSelfishChild(jp(target,args,result,children),receiver);
datatype  'a optional
  con data : 'a -> 'a optional
  con empty : 'a optional
datatype  'a jp
  con jp : 'a optional * 'a list * 'a optional * 'a jp list -> 'a jp
val getTarget = fn : 'a jp -> 'a optional
val getArgs = fn : 'a jp -> 'a list
val getResult = fn : 'a jp -> 'a optional
val getChildren = fn : 'a jp -> 'a jp list
val contains = fn : ''a list * ''a -> bool
val isChildOf = fn : ''a jp * ''a jp -> bool
val hasSelfishChild = fn : ''a jp * ''a optional -> bool
```

lation, moreover, does not just check OF, but can also check CF.

The motivation for a generic join point model is not just the formulation of LoD but also to gain a better insight into the connection between dynamic and lexical join point models. We use the generalized checker to identify potential imbalances in AspectJ which can handle DOF elegantly but which has difficulties with LCF.

3.2.1 *Mapping* OF *and* CF *to* JPF

We use JPF to check DOF as follows. Given a dynamic join point model, an execution trace can be expressed as a sequence of abstract join points in JPF, α being object ID. Join points are method invocations. The enclosing join point is the parent in the control flow; we apply the LoD pointcut to get DOF.

We use JPF to check LCF as follows. Given a lexical join point model, part of the abstract syntax tree of the program can be modeled as an abstract join point tree in JPF, α being class name. Join points are signatures of call sites. The enclosing join point is the signature of the method in which the call site resides. To run the aspect, a suitable ordering has to be given to the elements of children: all constructor calls, followed by local method calls, followed by the other join points; we apply the LoD pointcut to get LCF.

JPF is mapped into our implementations as follows. *Argument Rule* is mapped into the context sensitive category, in which α's are potential preferred suppliers only if they are in the context of the current method execution. For the *Associated Rule*, we map the result α's of local method calls and created α's in the current method execution (we call them *Locally Constructed*) to the context sensitive category for the same reason as the *Argument Rule*'s, while the direct part α's of the target of the current method execution are mapped to the context insensitive category, since those direct part α's are potential preferred in any method execution of their containing target.

3.3 Complexity of LoD checking

LCF and DOF predicates can be checked polynomially in the length of the programs and the length of the executions (number of join points in the dynamic call graph). LOF can be shown to be undecidable by a simple reduction to the halting problem. See Table 1 for a summary.

	class	object
lexical	LCF(polynomial)	LOF(undecidable)
dynamic	DCF(polynomial)	DOF(polynomial)

Table 1: Complexity of LoD predicates.

We wished we could check LoD using aspects similar to the the ones shown in Listings 1 and 3, which use the static **`declare warning`** mechanism of AspectJ. It is reasonable to expect that we cannot implement DOF in that way, since DOF needs to be checked dynamically. However, we can't implement LCF either, even though it is polynomial and can be checked statically.

The next two sections present our dynamic checkers in AspectJ for the object form (DOF) and the class form (DCF), respectively. Our DCF checker illustrates our argument about how AspectJ should be extended to provide more power for static checking.

4. CHECKING THE OBJECT FORM

There are some subtle differences between DOF and DCF. Before we describe our DOF and DCF implementations, we mention some general issues affecting the implementation.

- When checking whether an object is a potential preferred

supplier, we can use either reference semantics or value semantics. For DOF we use reference semantics.

- In DOF the state of the object is important, DOF is sensitive to the order of assignments to instance variables, since we want to always capture the most recently updated "direct parts" of "this" objects. In DCF, the order of assignments is irrelevant, since we only care about the types.
- In DOF we distinguish between *static* and *non-static* methods since the target objects are different, while in DCF this issue is irrelevant.

To keep our implementation as concise as possible, we relax the requirements as follows:

- We only check message-sends.
- We don't check the legality of any method calls residing in a static method definition, neither do we check the legality of the calls to a static method.
- We don't view objects contained in a `Collection` instance variable as "direct parts."

Listing 4 is a utility abstract class that defines all the pointcuts needed in this implementation (some of them are needed later in the class form checker). Those pointcuts pervasively touch programs and make extensive use of property-based pointcuts. The `scope()` pointcut prevents the aspects from advising the LoD checker code, which is generally desired to avoid circular advice. The `SelfCall` pointcut captures the method calls sent to **`this`** in a method or constructor execution. Other pointcuts are self-explanatory.

4.1 Implementation

The implementation uses three concrete aspects with one or two short advice each and a few auxiliary methods. The design of the implementation is clean and easy to understand due to the use of a dynamic join point model. Figure 2 shows the UML diagram of the object form checker.

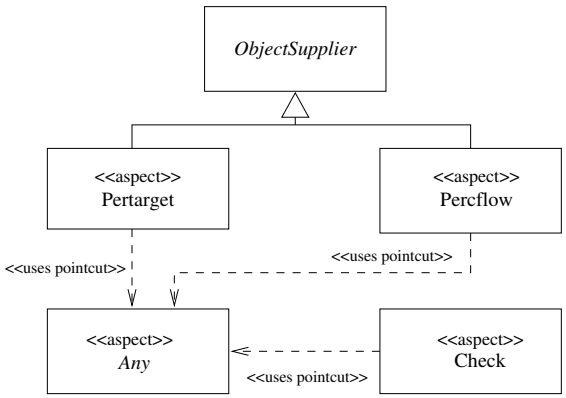


Figure 2: The UML diagram of the object form checker

There are two tasks that need to be performed. One is to collect all of the preferred supplier objects on which methods can be called from an object/context. The other is to verify that each method call makes valid calls on a preferred supplier object of the corresponding "this" object. There are two categories of preferred supplier objects for an object. The first category is context-insensitive: in a method execution on an object, it is legal to call a method on any instance variable of that object. The second category is context-sensitive in that some objects are only preferred in the scope of a method, for example, the method call on an argument object is only legal within the method body of the enclosing method.

Listing 4: Any.java

```
package lawOfDemeter;
public abstract class Any {
  public pointcut scope(): !within(lawOfDemeter..*)
    && !cflow(withincode(* lawOfDemeter..*(..)));
  public pointcut StaticInitialization(): scope()
    && staticinitialization(*);
  public pointcut MethodCallSite(): scope()
    && call(* *(..));
  public pointcut ConstructorCall(): scope()
    && call(*.new (..));
  public pointcut MethodExecution(): scope()
    && execution(* *(..));
  public pointcut ConstructorExecution(): scope()
    && execution(*.new (..));
  public pointcut Execution():
    ConstructorExecution() || MethodExecution();
  public pointcut MethodCall(Object thiz,
    Object target): MethodCallSite()
    && this(thiz)
    && target(target);
  public pointcut SelfCall(Object thiz,
    Object target): MethodCall(thiz,target)
    && if(thiz == target);
  public pointcut StaticCall(): scope()
    && call(static * *(..));
  public pointcut Set(Object value): scope()
    && set(* *.*) && args(value);
  public pointcut Initialization(): scope()
    && initialization(*.new(..));
}
```

Listing 5: ObjectSupplier.java

```
abstract class ObjectSupplier {
  protected boolean containsValue(Object supplier){
    return targets.containsValue(supplier);
  }
  protected void add(Object key,Object value){
    targets.put(key,value);
  }
  protected void addValue(Object supplier) {
    add(supplier,supplier);
  }
  protected void addAll(Object[] suppliers) {
    for(int i=0; i< suppliers.length; i++)
      addValue(suppliers[i]);
  }
  private IdentityHashMap targets =
    new IdentityHashMap();
}
```

The class `ObjectSupplier` defines a repository and a set of supporting methods for looking up and adding preferred supplier objects to an object so that its two subaspects can access them. Note that we use `java.util.IdentityHashMap` [4] class to implement the repository, since we use reference semantics to compare two objects.

[4] `java.util.IdentityHashMap` is a class available since JDK 1.4.

The aspect `Pertarget` implements the only context-insensitive preferred objects situation, i.e., instance variables of an object, by advising the **set** join point. It is declared as **pertarget**(`Any.Initialization()`) so that once a new object o is initialized, an aspect instance of `Pertarget` will be created and associated with o, and each aspect instance can correctly maintain the direct part relationship between the instance variables and their hosting object o. The before-advice on the **set** join point handles that logic, in which the `fieldIdentity` method is used so that if an object o_1 has been set as a direct part of an object o_2 through a field f, then later o_2's f is set to another object o_3, we can replace o_1 with o_3 and always maintain the correct direct part relationships.

Listing 6: Pertarget.java

```
public aspect Pertarget
  extends ObjectSupplier
  pertarget(Any.Initialization()) {
  before(Object value): Any.Set(value) {
    add(fieldIdentity(thisJoinPointStaticPart),
      value);
  }
  public boolean contains(Object target) {
    return super.containsValue(target) ||
      Percflow.aspectOf().containsValue(target);
  }
  private String fieldIdentity(JoinPoint.StaticPart
    sp) {
    String fieldName = sp.getSignature().
      getDeclaringType().getName() + ":" +
      sp.getSignature().getName();
    if(fieldNames.containsKey(fieldName))
      fieldName=(String)fieldNames.get(fieldName);
    else
      fieldNames.put(fieldName,fieldName);
    return fieldName;
  }
  private static HashMap fieldNames =
    new HashMap();
}
```

The aspect `Percflow` implements all the context-sensitive preferred objects situations, by advising `Any.Execution()` and examining results of `Any.SelfCall(Object,Object)`, `Any.StaticCall()`, or `Any.ConstructorCall()` to collect the corresponding preferred supplier objects. `Percflow` is declared as **percflow**(`Any.Execution()|| Any.Initialization()`) to simulate the execution scope of a method, instead of requiring manual stack operations. Note that here `Any.Initialization()` is necessary because in AspectJ an instance variable initialization defined outside any constructor definition is not in the execution of any constructor.

Listing 7: Percflow.java

```
aspect Percflow extends ObjectSupplier
  percflow(Any.Execution()|| Any.Initialization()){
  before(): Any.Execution() {
     addValue(thisJoinPoint.getThis());
     addAll(thisJoinPoint.getArgs());
  }
  after() returning (Object result):
    Any.SelfCall(Object,Object) || Any.StaticCall()
    || Any.ConstructorCall() {
      addValue(result);
  }
}
```

The checking logic happens in the `Check` aspect, which defines the after-advice on method call join points and checks whether a target is a preferred supplier according to LoD.

Any style rule has exceptions, including LoD. To make the checker be practically useful, the method calls on some specific objects should be allowed in any situation, e.g., *System.out.println(...)* should be allowed to be called anywhere. The `IgnoreTargets` pointcut defines this logic by capturing all those kinds of objects, whose domain currently includes all the public static variables declared in the classes in the packages beginning with `java`. We don't want to check method calls on some stable types either, so we use pointcut `IgnoreCalls` to list those method calls. The `Check` aspect uses the two pointcuts to ignore checking in those two situations. Users can always change those domains by customizing the pointcuts.

Listing 8: Check.java

```
aspect Check {
  private pointcut IgnoreCalls():
    call(* java..*.*(..));
  private pointcut IgnoreTargets():
    get(static * java..*.*);
  after() returning(Object o):IgnoreTargets() {
    ignoredTargets.put(o,o);
  }
  after(Object thiz,Object target):
    Any.MethodCall(thiz,target)
    && !IgnoreCalls() {
    if (!ignoredTargets.containsKey(target) &&
      !Pertarget.aspectOf(thiz).contains(target))
       System.out.println(
         " !! LoD Object Violation !! "
         + thisJoinPointStaticPart);
  }
  private IdentityHashMap
   ignoredTargets = new IdentityHashMap();
}
```

5. CHECKING THE CLASS FORM

The class form checker has the same functional architecture as the object form checker in that both of them use suppliers and a checker that acts as the client of the suppliers. But from the design point of view, our class form checker uses a different AOP framework from the object form checker's, in which, for each subrule, we have corresponding advice. In the class form checker, we have used abstract aspects to specify that when some interesting scenario happens some advice will be executed. The concrete subaspects reuse the advice defined in the superaspect by concretizing the interesting scenario. Of course, the concrete subaspects can customize the process logic for their scenarios by overriding abstract methods. Figure 3 shows the UML diagram of the class form checker.

Listing 9: ClassSupplier.java

```
abstract class ClassSupplier {
  protected abstract List
  getSuppliers(JoinPoint.StaticPart enclosingjsp,
    JoinPoint.StaticPart jsp);
}
```

The classes and aspects: `ClassSupplier`, `Pertype`, `Perscope`, and `Check` make up an aspect-oriented framework which defines the generic checking behavior. This is a reusable aspect framework and for different versions of LoD we can add different sets of subaspects.

We have implemented `Pertype` and `Perscope` as abstract aspects, each of which defines an abstract pointcut (with the same name as the aspect) which is used to collect preferred supplier types. Similar to the situations in the object form checker, the two abstract as-

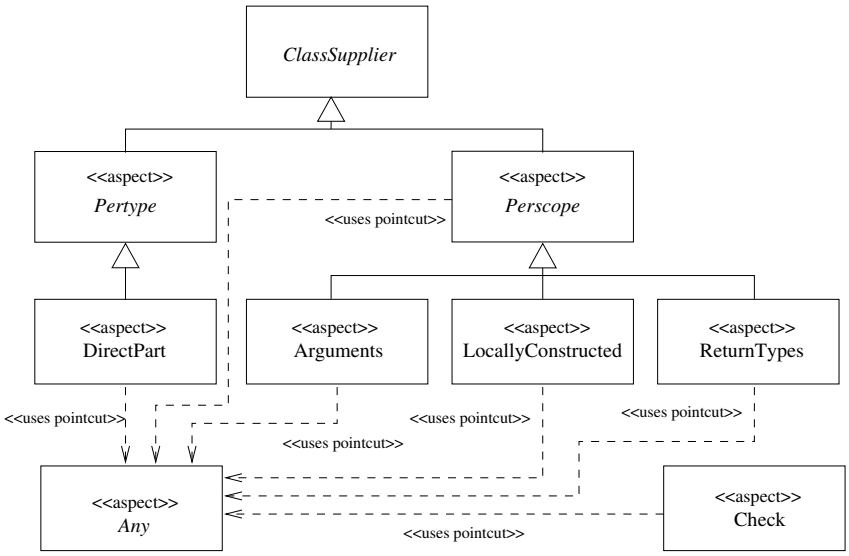


Figure 3: *The UML diagram of the class form checker*

Listing 10: Pertype.java

```
abstract aspect Pertype extends ClassSupplier {
  abstract pointcut Pertype();
  before(): Pertype() {
    targets.put(thisJoinPointStaticPart.
      getSignature().getDeclaringType(),
      getSuppliers(thisEnclosingJoinPointStaticPart,
        thisJoinPointStaticPart));
  }
  protected static boolean contains(Class thisType,
    Class targetType) {
    if(targets.containsKey(thisType)) {
      List alloweds = (List)targets.get(thisType);
      Iterator it=alloweds.iterator();
      while(it.hasNext()) {
        if(targetType==it.next())
          return true;
      }
    }
    return Perscope.contains(targetType);
  }
  private static HashMap targets = new HashMap();
}
```

Listing 11: Perscope.java

```
abstract aspect Perscope extends ClassSupplier {
  abstract pointcut Perscope();
  before() : Any.Execution() {
    st.push(new HashSet());
  }
  before() : Perscope() {
    HashSet aSet = (HashSet) st.peek();
    aSet.addAll(getSuppliers(
      thisEnclosingJoinPointStaticPart,
      thisJoinPointStaticPart));
  }
  after(): Any.Execution() {
    st.pop();
  }
  static boolean contains(Class targetType) {
    HashSet innermost = (HashSet)Perscope.st.peek();
    return innermost.contains(targetType);
  }
  private static Stack st = new Stack();
}
```

pects correspond to the two different situations in which the types are preferred. Table 2 lists the correspondences between the two checkers.

aspect	object	class
context-insensitive	Pertarget	Pertype and subaspect
context-sensitive	Percflow	Perscope and subaspects

Table 2: The correspondences between object/class form checkers.

The first situation is the context-insensitive situation as defined by `Pertype`, in which some types are always preferred for a given type. The only context-insensitive situation is the direct part situation, where the types of the instance variables (including inherited instance variables) of a class are always preferred in any methods of the class. The second situation is the context-sensitive situation as defined by `Perscope`, in which the types are only preferred when the call sites are in the stack of a particular method execution. (An example of that situation is the arguments situation, where the types of arguments are only legal for the scope of the method body.) All of the concrete aspects extending any of the abstract aspects are supposed to give:

Listing 12: Check.java

```
aspect Check {
  private pointcut IgnoreCalls():
    call(* java..*.*(..));
  after(): Any.MethodCallSite() && !IgnoreCalls() {
    Class targetType = thisJoinPointStaticPart.
        getSignature().getDeclaringType();
    Class thisType =
      thisEnclosingJoinPointStaticPart.
        getSignature().getDeclaringType();
    if(!Pertype.contains(thisType,targetType)
      && !targetType.isAssignableFrom(thisType))
      System.out.println(
        " !! LoD Class Violation !! "
        + thisJoinPointStaticPart);
  }
```

Listing 13: DirectPart.java

```
aspect DirectPart extends Pertype {
  public pointcut Pertype():
    Any.StaticInitialization();
  protected List getSuppliers(JoinPoint.StaticPart
    ejsp,JoinPoint.StaticPart jsp) {
    List suppliers=new ArrayList();
    Class currentClass =
      jsp.getSignature().getDeclaringType();
    while(currentClass != null) {
      Field[] fields =
        currentClass.getDeclaredFields();
      for(int i=0; i<fields.length; i++)
        suppliers.add(fields[i].getType());
      currentClass=currentClass.getSuperclass();
    }
    return suppliers;
  }
}
```

Listing 14: Arguments.java

```
aspect Arguments extends Perscope {
  pointcut Perscope(): Any.MethodExecution()
    || Any.ConstructorExecution();
  protected List
  getSuppliers(JoinPoint.StaticPart ejsp,
  JoinPoint.StaticPart jsp) {
    Class thisClass =
      jsp.getSignature().getDeclaringType();
    List parameterTypes = new ArrayList();
    parameterTypes.add(thisClass);
    parameterTypes.addAll(
      Arrays.asList(((CodeSignature)jsp.
      getSignature()).getParameterTypes()));
    return parameterTypes;
  }
}
```

Listing 15: LocallyConstructed.java

```
aspect LocallyConstructed extends Perscope {
  pointcut Perscope():
    Any.ConstructorCall();
  protected List getSuppliers(JoinPoint.StaticPart
    ejsp,JoinPoint.StaticPart jsp) {
    List supplier = new ArrayList();
    supplier.add(jsp.getSignature().
      getDeclaringType());
    return supplier;
  }
}
```

Listing 16: ReturnTypes.java

```
aspect ReturnTypes extends Perscope {
  pointcut Perscope(): Any.MethodCallSite();
  protected List
  getSuppliers(JoinPoint.StaticPart ejsp,
    JoinPoint.StaticPart jsp) {
    List supplier = new ArrayList();
    if(ejsp.getSignature().getDeclaringType()!=
      jsp.getSignature().getDeclaringType())
      return supplier;
     supplier.add(((MethodSignature)jsp.
       getSignature()).getReturnType());
     return supplier;
  }
}
```

- a definition of the corresponding abstract pointcut to concretize where the advice defined in the superaspect should happen;
- an implementation of the abstract method `getSuppliers` declared in class `ClassSupplier` to expose the preferred types for its particular scenario.

There are four concrete aspects extending `Pertype` or `Perscope` aspect, which correspond to the four rules of LoD (we view the associated rule to be three rules) respectively. To make our checker practically useful, we allow method calls on super types in the method execution of their subtypes and as in the object form checker, we also allow exceptions to the class form of the LoD, which is defined and configurable by pointcut `Check.IgnoreCalls()`. The `Check` aspect does the straightforward checking logic.

This implementation is a dynamic checker for LCF. However, in Listings 9 through 16, we only use static type information of classes or methods (we use Java Reflection [6] to get the types of the instance variables of a class, but this information can also be easily available at compile time [12]) and all the advice are defined on statically determinable pointcuts, hence it is natural to argue that AspectJ should have been able to support static checking for LCF. What is missing is some sort of statically executable mechanism that would permit these advice to be part of **`declare`** statements, which can be defined by users and executed at compile time so that users can write more complex logic than that of **`declare error`** or **`declare warning`**.

In the next section, we propose two possible extensions to AspectJ to make the above scenario a reality.

6. EXTENDING ASPECTJ

As discussed earlier, the checking of the class form of the LoD would be possible if AspectJ supported a mechanism for defining some logic that is more complex than that of **`declare error`** or **`declare warning`**, but is still statically executable. Here is our argument why one would expect this kind of feature from AspectJ.

1. The **`declare`** mechanism is a very useful feature of AspectJ for supporting simple checking at compile time. It would be a natural extension for AspectJ to support advice on those statically determinable pointcuts and get those advice executed at compile time.
2. AspectJ already makes rich static information about the program available through **`thisJoinPointStaticPart`**, but currently this information is only accessible to regular run time advice. Making this static information accessible to advice executed at compile time, would allow many interesting properties about a program (like LCF) to be checked at compile-time.
3. AspectJ is expressive enough to capture interesting join points in the base program structure. We would like to make use of the expressiveness of the pointcut language by lifting some of the current restrictions on statically determinable pointcut definitions.

The following two sections describe our two visions about how AspectJ can be extended to better support static checking approach.

6.1 Statically Executable Advice

Statically executable advice is a feature that would allow AspectJ to support advice on statically determinable pointcuts and to execute the advice at compile-time. Here is our proposed architecture.

First, a new interface `StaticallyExecutableAdvice` needs to be added to AspectJ's API, so that users who want to use this feature can define their computation by implementing this interface (Listing 17).

Listing 17: StaticallyExecutableAdvice.java

```
public interface StaticallyExecutableAdvice {
  void beforeCall(StaticJoinPoint sjp);
  void afterCall(StaticJoinPoint sjp);
  void beforeExecution(StaticJoinPoint sjp);
  void afterExecution(StaticJoinPoint sjp);
  void beforeGet(StaticJoinPoint sjp);
  void afterGet(StaticJoinPoint sjp);
  //There will be more for other join points ...
}
```

The type `StaticJoinPoint` will be added into AspectJ's API to provide static information collected during the compilation process, similar to **`thisJoinPointStaticPart`** in the current implementation of AspectJ.

One can use the statically executable advice feature by implementing the `StaticallyExecutableAdvice` interface, which can access the static information about the join point through the argument `StaticJoinPoint` and do the static analysis. Users can associate their statically executable advice only with the statically determinable pointcuts in aspects through a new special **`declare`** mechanism called **`declare advice`**.

Listing 18 shows how statically executable advice would be used, where `StaticallyExecutableAdviceImpl` is the name of an exemplar class implementing the `StaticallyExecutableAdvice` interface.

Listing 18: Foo.java

```
aspect Foo {
  declare advice: execution(* A.test(..)) ||
    call(* B.foo()):
      StaticallyExecutableAdviceImpl;
}
```

When AspectJ's compiler compiles aspect `Foo` and reaches the declaration, the `StaticallyExecutableAdviceImpl` class is dynamically loaded and is used to create an instance (for each of such classes, there is typically only one singleton instance). Then when AspectJ's compiler reaches any join point that matches the pointcut definitions, the compiler can construct the corresponding `StaticJoinPoint` object `sjp` and call the corresponding methods specified by `StaticallyExecutableAdvice` interface on that instance of `StaticallyExecutableAdviceImpl` with `sjp` bound to the argument. Those statically executable advice get executed at compile time. The before and after prefixes in the interface methods have the same semantics as before and after advice except now it is happening at compile time.

6.2 Meta-Level Advice

Compile-time analysis can also be achieved by another, more general approach, which uses meta level advice.

AspectJ uses a dynamic join point model where the join points are points in the execution of a program. AspectJ has excellent support for the dynamic join point model, but only limited support for the corresponding lexical join point model. Through the dynamic join points we can access the corresponding lexical join points. For example, in the dynamic join point model we can easily talk about all method calls happening during the execution of a program by writing **`call`**`(* *(..))`. However, we cannot easily express in AspectJ the set of all call-site signatures contained in a given program by using a notation like: `Shadow(`**`call`**`(* *(..)))`.

An alternative to extending AspectJ with statically executable advice is to expose the meta model of an AspectJ program in the form of a class graph and an abstract syntax tree and to offer navigational capabilities in the meta model commensurate with the navigational capabilities in the dynamic model.

One justification why AspectJ should not expose the meta model is that we could just use Java to parse an AspectJ program and process the abstract syntax tree to check something like LCF. But a parser for AspectJ is already available in the AspectJ's compiler ready to be reused. Another justification why AspectJ should not expose the meta model is that the static join points don't need enhancement (advice) as opposed to the dynamic join points that need checking logic to be added. But the meta-level advice can be viewed as a base-level advice and we would like to use all the capabilities of AspectJ to process the meta-level objects (abstract syntax trees).

We propose that AspectJ expose its meta-level objects and add navigational support on those objects so that users can easily implement static checking like LCF checking by just traversing the AST. The Demeter traversal specifications [16] would be useful as navigational support. For example, we could then write "from Program to CallSite" instead of `Shadow(`**`call`**`(* *(..))`. This kind of navigational support is also very useful at the base level [18].

7. CONCLUSION

The paper makes the following contributions:

1. We define a generic join point model for checking LoD generically. The join points are join point trees, the pointcuts are predicates over the join points, and advice is statically checked as in the **`declare warning`** mechanism in AspectJ.

2. We show how the generic LoD checker is used to check DOF, and we provide an elegant implementation in AspectJ.

3. We show how the generic LoD checker is used to check LCF, and we note that we cannot provide an elegant implementation in AspectJ. We can only provide an approximation to LCF which checks DCF (per a particular execution).

4. AspectJ provides lexical join point information through dynamic join points. The paper proposes two ways to extend AspectJ to provide access to lexical join points directly.

- The first proposes statically executable advice and pointcut designators over lexical join points.
- The second proposes statically executable meta-advice over the exposed abstract syntax tree of the program, and applies aspect-oriented programming (including Demeter style support for traversal-related concerns) to the abstract syntax tree.

This paper concludes a line of research in aspect-oriented exploration. The Law of Demeter, postulated some 15 years ago, reduces coupling in OO programs at the expense of an increase in scattering and tangling. Adaptive and AOP techniques have been explored to deal with this inherent crosscutting resulting from following LoD. In this paper we go back and use a general-purpose AOP language to implement LoD style checkers in a non-intrusive, extensible, convenient way.

Acknowledgments

We are grateful to Sergei Kojarski for evaluating the different LoD checkers and for helping integrate the different versions. Many thanks to Paul Freeman for developing an industrial quality LoD checking tool, and for his feedback on our LoD checkers.[5] We thank Doug Orleans, Johan Ovlinger, Yi Qian, Fabio Rojas, Theo Skotiniotis and the anonymous reviewers for their valuable feedback.

8. REFERENCES

[1] J. Aldrich, C. Chambers, and D. Notkin. Architectural reasoning in ArchJava. In B. Magnusson, editor, *Proceedings of the 16th European Conference on Object-Oriented Programming*, number 2374 in Lecture Notes in Computer Science, pages 334–367, Málaga, Spain, June10-14 2002. ECOOP 2002, Springer Verlag.

[2] AOSD 2002. *Proceedings of the 1st international conference on Aspect-Oriented Software Development*, Enschede, The Netherlands, Apr. 2002. ACM Press.

[3] V. R. Basili, L. C. Briand, and W. L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22(10):751–761, 1996.

[4] M. Deters and R. K. Cytron. Introduction of program instrumentation using aspects. In *Proceedings of the OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, Tampa, FL, Oct. 2001. ACM.

[5] P. Freeman, S. Kojarski, K. Lieberherr, D. Lorenz, and P. Wu. Aspect-Oriented Design and Implementation of a Law of Demeter Checking Tool. Technical Report NU-CCS-03-01, Northeastern University, Jan. 2003.

[6] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java Language Specification*. Addison-Wesley, 2 edition, 2000.

[7] A. Hunt and D. Thomas. *The Pragmatic Programmer*. Addison-Wesley, 2000.

[8] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *Proceedings of the 15th European Conference on Object-Oriented Programming*, number 2072 in Lecture Notes in Computer Science, pages 327–353, Budapest, Hungary, June 18-22 2001. ECOOP 2001, Springer Verlag.

[5]The source code for the checkers is available from the Demeter home page: `http://www.ccs.neu.edu/research/demeter/`, in the directory `demeter-method/LawOfDemeter/AspectJCheckers/`.

[9] K. J. Lieberherr and I. Holland. Assuring good style for object-oriented programs. *IEEE Software*, pages 38–48, September 1989.

[10] K. J. Lieberherr and I. Holland. Formulations and Benefits of the Law of Demeter. *SIGPLAN Notices*, 24(3):67–78, March 1989.

[11] C. V. Lopes and G. Kiczales. Recent developments in AspectJ. In S. Demeyer and J. Bosch, editors, *Object-Oriented Technology. ECOOP'98 Workshop Reader*, number 1543 in Lecture Notes in Computer Science, pages 398–401. Workshop Proceedings, Brussels, Belgium, Springer Verlag, July 20-24 1998.

[12] D. H. Lorenz and J. Vlissides. Pluggable reflection: Decoupling meta-interface and implementation. Technical Report NU-CCS-02-10, College of Computer and Information Science, Northeastern University, Boston, MA 02115, Sept. 2002. To appear in International Conference on Software Engineering, 2003.

[13] H. Masuhara, G. Kiczales, and C. Dutchyn. Compilation semantics of aspect-oriented programs. In R. Cytron and G. Leavens, editors, *Foundations of Aspect-Oriented Languages Workshop*, pages 17–26, Enschede, Netherlands, 2002.

[14] N. Minsky and P. Pal. Imposing The Law of Demeter and Its Variations. In *TOOLS Conference*, Santa Barbara, CA, 1996.

[15] D. Orleans. Incremental programming with extensible decisions. In AOSD 2002 [2].

[16] J. Palsberg, B. Patt-Shamir, and K. Lieberherr. A new approach to compiling adaptive programs. *Science of Computer Programming*, 29(3):303–326, 1997.

[17] M. Shomrat and A. Yehudai. Obvious or not?: regulating architectural decisions using aspect-oriented programming. In AOSD 2002 [2], pages 3–9.

[18] J. Sung. Aspectual Concepts. Master's thesis, Northeastern University, June 2002. Technical Report NU-CCS-02-06.