

Controlling the Complexity of Software Designs

Karl J. Lieberherr, College of Computer and Information Science,
Center for Software Sciences, Northeastern University, Boston, MA 02115

Abstract

Our research has focused on identifying techniques to develop software that is amenable to refactoring and change. The Law of Demeter (LoD) was one contribution in this effort. But it led to other problems. With the current state of the art focused on Aspect-Oriented Software Development (AOSD), it is useful to revisit the general objectives of the LoD and adapt it to the new ideas. Hence we introduce the Law of Demeter for Concerns and discuss the important intersection of these approaches with traversals. We explore the ramifications of the Laws of Demeter (LoD and LoDC) to achieve better separation of concerns through improved software processes. They are supported by language mechanisms that are implemented using novel applications of automata theory.

A revised version of this paper and the slides are available from the Demeter website:

*<http://www.ccs.neu.edu/research/demeter>
in the directory `papers/icse-04-keynote/`*

1. Introduction

Our research has focused on identifying techniques to develop software that is amenable to refactoring and change. The Law of Demeter (LoD) [30, 29, 16, 7] was one contribution to this effort. The LoD, *talk only to your friends* in the popular formulation, resulted in problems when followed in a strictly object-oriented context and it was these problems that sent us towards our investigation of AOSD.

In the popular formulation of the LoD, there are many ways to interpret “friends” and “talk”. Friends could be objects or classes or components and talking can mean “refer to” or “call”. In the most general form, we have a set of modules (e.g., methods and objects) and each module talks only to closely related modules (the friends). The goal is that a module becomes more understandable if it does not refer to too many other modules, but instead deals with only the appropriate modules that are needed to achieve a purpose. The LoD helps to control information overload for

the designer. The LoD goes beyond encapsulation by making minimal use of public knowledge (interfaces). The LoD is an application of the Low Coupling Principle by making the notion of bad coupling explicit and checkable by tools.

We limit our focus to object-oriented systems with classes and methods and we interpret the LoD in this context where it was also first observed. The reason is not only due to the popularity of object-oriented systems, but also due to its inherent nature to model each entity as a module and the whole system as a relation between its modules. This relation can then be modeled as an interesting graph structure, which we will use in our discussion. The original formulation of the LoD was: All methods may only send messages to preferred supplier objects. A method’s preferred supplier objects are its “friends” and are the immediate part objects, the argument objects, including “this” and objects that are created directly in the method. Later, we generalized the notion of preferred supplier objects to: the immediate part objects (both the stored and the computed parts) based on a suggestion by Walter Hürsch. A computed part object is one that is returned by a message sent to “this”.

2. LoDC

From the early days of what we called Adaptive Programming[26], we implicitly used a stronger form that we call here the Law of Demeter for Concerns (LoDC). In the LoDC, we restrict the friends further: only friends that contribute to a common set of concerns. So the LoDC is: *Talk only to your friends that contribute to a common set of concerns* or *that share the same concerns*. There may be multiple concerns in play and each communication should be restricted to those preferred supplier objects that contribute to those concerns. A similar view is taken by Colyer at al. [5] with the Principle of Dependency Alignment.

We cannot avoid talking to friends that don’t contribute to the current concerns but we want to impose this communication using a separate module. This imposition has the advantage that the code for the current concerns stays unpolluted and does not get cluttered with calls related to foreign concerns. This makes code more reusable and flexible.

Consider a base application that implements a set of concerns and now we want to log some of the method calls in the base application. In other words, we want to implement the logging concern that has been often used in the AOSD community. The power of AOSD is not well represented by this example but it suffices to illustrate the connections we are after. The logging concern is distinct from the application's concerns; indeed it is an intrusive concern that wants to watch what the base application does. The logging concern wants to send messages to a logfile with information from the base application.

Instead of adding the logging calls in the base application code, violating the LoDC (the “that contribute to a common set of concerns” part), we need to enhance the base application from the “outside”. We need to find a convenient way to specify the places in the program execution where those additional calls to the logfile object should happen. The logging concern is an example of a concern that cuts across the base program.

By following the LoDC, we build an interesting abstraction in the spirit of the guideline on creating abstractions: Form an abstraction instead of copying and modifying a piece of a program (e.g., [17]). Here we don't want to duplicate the log calls. What is interesting is that the aspectual abstraction cuts across several methods and classes.

The LoDC implies the LoD. The LoDC makes us think carefully in terms of concerns that are worthwhile to separate [12, 53, 21]. By a concern we mean any issue that the designer/programmer is concerned with, e.g., the synchronization policy of a system, the implementation of a use case, or the exception handling policy of a system. The LoDC promotes layered software development where each layer implements one or more concerns[4]. Each layer has a set of extension points to which the next set of concerns may connect. The extension points are defined outside the current layer, using a “coordinate system” that exists in the current concerns. This mechanism allows concerns, whose nature requires them to interplay in complex and systematic ways with other concerns (crosscutting concerns), to mix with each other at well specified and publicly available points. These points, being defined outside of a layer, allow for the non-invasive addition/extension of the system through the definition and incorporation of new concerns. Alternatively when concerns are crosscutting, their ad hoc implementation would be scattered resulting in their definition being tangled with other concerns. The LoDC has a subtle influence on how we structure our software. If a new concern has to be added from the outside through a set of extension points, we will be careful to make it easy to express the coordinates of the extension points.

Let's consider a program that implements a set of concerns C_s . Now we add a new concern C_{new} . Consider C_s and C_{new} implemented and consider an execution of

the program. The set of calls that cross the boundary from within C_s to C_{new} are the sets of calls (extension points) we would like to capture in a separate abstraction. And the code executed after those calls until we return back to originating calls belonging to C_s we want to capture as a separate enhancement to the code belonging to C_s . The implementation of C_{new} is an “outside” abstraction to be added to the program to satisfy the “that contribute to a common set of concerns” part. The “talk only to your friends” part takes care of organizing the code inside C_s .

In many situations we might be able to partition the objects based on concerns. In that case, the LoDC disallows calling methods of objects that (1) are not friends and that (2) don't contribute to the current concerns. In some cases, C_{new} fits into one object but in many cases it cuts across several objects.

Dealing with non-invasive extensions for crosscutting concerns is the theme of AOSD (Aspect-Oriented Software Development) [12]. For a history of Aspect-Oriented Programming (AOP), see [40]. Composition filters [2] were an early technique to extend a class non-invasively using layers of composable filters. Ivar Jacobson [22] has proposed aspect-oriented ideas but without using an expressive notation to describe sets of extension points.

2.1. Extension languages

What do we gain from factoring out the logging calls? We gain the advantage that we can easily plug-in and plug-out the logging aspect. But we don't gain much, if we have to enumerate one by one the places where the calls should happen. In the terminology of Filman et al. [13] we would say that we have obliviousness without quantification. It could be that writing down the “coordinates” of the calls is more cumbersome than writing the calls themselves. But in practice we often find that the coordinates follow a pattern that is easy to express. For example, we might want to log all method calls of the form `C.withdraw(..)`. (`..` means any number of arguments.) In this case, factoring out the logging concern is a big win. We are guaranteed to capture all those method calls while the manual solution might miss some. AspectJ solves the problem as follows:

```
aspect Logging {
    pointcut find_Cwithdraw():
        call(* C.withdraw(..));
    after find_Cwithdraw() {
        System.out.println(thisJoinPoint);}
}
```

This program makes the assumption that we have not accidentally called an unrelated method `withdraw` that has nothing to do with `withdrawal`.

We need techniques to state the coordinates semantically. When you agree on a coordinate with your friend to meet at the main railway station, you might say: Let's meet at the

main meeting point marked by arrows. When she asks for detailed directions you might hesitate because you are not sure you remember the details. When you tell her to turn left at the Swissair office, you will be in trouble because it no longer exists. It could also be that the main meeting point has been moved but the high level description still holds.

Furthermore, manually adding log calls will increase the amount of information that is found in the original module. This intermixing in a module's operations (tangling) increases the code base of the module while decreasing the ability to reason about what is the functionality of this module. Iterative manual additions of other concerns add to the severity of this issue.

However, we also have to remember that when we modify the program and add more method calls of the form `C.withdraw(..)` that all those calls will also be automatically logged. We call this the adaptation dilemma for which solutions are emerging. See section 3.1.

What kind of extension language should we use? We often need to refer to large sets of coordinates (extension points) that have an internal structure to them and we need to describe the extension to be applied at each extension point (see [42] about better referential mechanisms in programming languages). At one extreme, we use a special purpose approach where the language is specialized to the concern. For example, if the concern at hand is synchronization, we use a language specialized to synchronization [19, 24, 41, 43]. If the concern at hand is traversal-based collaboration, we use a language specialized for traversals and a language to express the collaboration that happens during the traversal [26]. Other examples of concern-specific languages are discussed by Duzan et al. [11]. Concern-specific approaches have many advantages but they lead to many small, collaborating languages and for their implementation they use similar mechanisms. Therefore, it has been very fruitful to use general-purpose approaches of which AspectJ [23] and HyperJ [20] are the most prominent ones. Also predicate dispatch languages are close to general purpose aspect languages [48]. Context objects are also an early form of general purpose aspects [55].

In section 3 we examine commonalities between following the LoD and LoDC. In section 4 we investigate complex request interfaces as a useful view to address wide interfaces caused by following the LoD. In section 5 we discuss applications of the LoD to domains other than writing methods. In section 6 we discuss tools for following the LoD. Section 7 touches on integrating aspects and components. Section 8 considers challenges and open problems and section 9 concludes the paper.

3. Following LoD and LoDC

The LoD is about organizing software within a set of concerns and the LoDC promotes the separation of a new concern if it is considered to be separable.

The LoD leads us into a dilemma (LoD dilemma): if we follow it, we have to write many small methods that duplicate information and if we don't follow it we get code that is difficult to maintain [27]. Traversal strategies [37] are the way out of this situation combined with either a traversal strategy interpreter (e.g., DJ) or a code generator (e.g., DAJ [31, 10], DemeterJ [35]).

3.1. Adaptation Dilemma

We continue with the logging example. The specification: `find_Cwithdraw` = "find all calls of the form `C.withdraw(..)`" is adaptive in that it can be "used" with many different base programs. But adaptation has its price: when the base program changes, we need to retest the specification to prove that the right elements are still being selected. We define the **adaptation dilemma** as follows:

When a parameterized program abstraction $P(Q)$ is given with a broad definition of the domain of the allowed actual parameters, we need to retest and possibly change the abstraction P when we modify the actual parameter, i.e., we move from $P(Q1)$ to $P(Q2)$.

In the above example, P is the specification `find_Cwithdraw` and Q is the base program. In the context of the adaptation dilemma, the abstraction P will change its meaning based on the actual parameter. It will adapt to the parameter but without referring in the definition of P to the details of the parameter. As we will see later, the adaptation dilemma also appears in Adaptive Programming (AP) from where it got its name [27]. Recently, it was also noticed by others [3, 6].

The high-level program abstractions used in AOP (called AOP abstractions) are different than "traditional" abstractions because of the analogous adaptation they cause. AOP practitioners using tools such as AspectJ, AspectWerkz, JBoss-AOP, JAC, DemeterJ etc. (see <http://www.aosd.net>) are happy to work with AOP abstractions by using good tool support. One reason is that AOP abstractions produce a lot of code that would be (1) tedious and error-prone to write by hand and (2) the code would be scattered over many methods and not pluggable. Instead of labeling AOP abstractions as wrong or breaking modularity, it is much better to find good ways of working with them.

Jonathan Aldrich [3] defines a module interface for the base program and if the implementation changes without affecting this interface, Open Modules guarantee that one

never has to retest the logging specification when the base program changes. This solution sacrifices some obliviousness [13]. Finding new ways of defining interfaces between base programs and aspects is an important problem.

3.2. Summing

Now consider a company object for a company consisting of divisions, departments and workgroups and each workgroup consists of employees, each one having a Salary-object. Consider the task of summing all those Salary-objects to compute the total salary paid by the company. Notice how this situation is similar to the logging example. In the logging example we had to find all calls of the form `C.withdraw(..)` and here we have to find all Salary-objects. In both cases we have a graph in which we need to select certain nodes and apply a method call at those nodes. The difference is that in one case it is a dynamic call graph (tree) and in the other an object graph. The adaptation dilemma also shows up here: when the class graph changes we need to retest whether the specification: “find all Salary-objects” is still correct or whether we have to constrain it further.

How do we express the coordinates of the salary objects? There is a danger that we violate the LoD. The disadvantage of such a violation is that we violate the DRY principle (Don’t Repeat Yourself, [17]) in that we duplicate information from the class diagram in the method body.

There is a good solution to the problem by mimicking the logging example. The method that adds the Salary-objects does not really have to know about divisions, departments and work groups. After all, we might want to use the summing method on a start-up company that only consists of one workgroup. So we use a specification: “visit all Salary-objects reachable from the Company-object”. There is a small problem as in the logging example: when we modify the Company-object we have to remember that all Salary-objects will be added to the total salary (adaptation dilemma).

With Java and the DJ library [49], the summing problem is solved by using the idioms from Figure 1. The classGraph is the class graph of the company (be it a start-up or a large corporation), whereToGo is “from Company to Salary” and whenAndWhatToDo is an instance of the SummingVisitor class (Figure 2). whereToGo is a graph with nodes and edges describing the plan for our journey. In this case it is a graph with one edge. The signatures of the visitor methods describe the “when” part and the bodies describe the “what” part of whenAndWhatToDo.

```
classGraph = new ClassGraph();
objectGraph = Source.
    parse("object description");

classGraph.traverse(
    objectGraph,
    whereToGo,
    whenAndWhatToDo);
```

Figure 1. Demeter Idiom for Java using DJ

```
class SummingVisitor extends
    edu.neu.ccs.demeter.dj.Visitor {
    int total;
    public void start() { total=0; }
    public void before(Salary o) {
        total += o.get_value(); }
    public Object getReturnValue() {
        return new Integer(total); } }
```

Figure 2. Summing Visitor using DJ (whenAndWhatToDo)

3.3. Commonalities

What is the abstract structure behind the logging and summing example? First we note a difference between the two. In the logging example the machine that runs the program will traverse the dynamic call graph. In the summing example, we can choose how to traverse the object graph and find all Salary-objects. I see a two-level graph structure with a selector language on top as the common structure between both situations [36, 37]. The first graph structure represents trees from which we want to select nodes and edges. The second graph structure represents meta-information about the trees. We use this graph structure to formulate sentences in the selector language to select nodes and edges in the first structure. The motto is to think statically, but act dynamically. The node selector expressions are formulated in terms of the static meta-information but they act on a dynamic graph.

The meta-information in the Summing example is a very well known entity: it is a class graph (in UML terminology called a class diagram). The class graph is used to decide which edges in the object graph are fruitful to traverse. For example, the Company-object might have an inventory part describing the structure of the company’s large inventory but not containing any Salary-object. Therefore there is no need to traverse into the inventory part and we can determine this statically based on the meta-information in the

class graph. Notice that the traversal is used to enumerate efficiently all the coordinates where a Salary-object will be found [52].

The meta-information in the Logging example is the program or an abstraction of the program. If we want to look for all method calls of a method `C.withdraw(...)`, we don't need to check all method calls dynamically. Instead, we can "mark" the calls of interest in the program. In other situations, we have to do some dynamic checking but the meta information helps to reduce the dynamic checking [45, 60]. In this case we don't need a traversal to enumerate the coordinates because the "machine" provides a default traversal [57]. We only need to efficiently recognize when we are at the right points in the dynamic execution.

The summing example also exhibits an application of the LoDC. The summing problem consists of three concerns: a traversal concern (`whereToGo`), an object structure concern (`classGraph` and `objectGraph`) and a collaboration concern (`whenAndWhatToDo`) and we would like to separate the three. The object structure concern consists of a static part (`classGraph`) and a dynamic part (`objectGraph`). The `whenAndWhatToDo` concern expresses what needs to be done on top of the traversal expressed by the `whereToGo` concern. The `whenAndWhatToDo` concern is naturally implemented using a visitor object [27, 49]. The signatures of the visitor methods express where the traversal needs to be enhanced and the bodies of the visitor methods express what code to execute at the enhancement points. Using Java and the DJ library the weaving of the three concerns is expressed in Figure 1. Notice the robustness of this code to changes of the class graph. `whenAndWhatToDo` might not need updating at all and `whereToGo` might need only minimal updating. Structure-shyness specifically and concern-shyness in general, are important goals of the Demeter project.

Another commonality between logging and summing is that we want to be shy with respect to the meta information (class graph and program implementation). The rule in [3] that prohibits external aspects from advising the internal calls of a module can be viewed as ensuring that external aspects are "implementation-shy" with respect to the internals of a module—thus allowing that implementation to change without affecting the external aspects. Open modules provide a very strict form of implementation-shyness at the expense of giving up some obliviousness.

For another commonality, consider the Logging example again. We are interested in all calls `C.withdraw(...)`. We can view the result as a subtree with the root a call to `main()` and the inner nodes all leading eventually to a `C.withdraw(...)` call. This situation is very analogous to the Summing example where we are interested in the sub tree of the company tree that has Salary-objects at its leaves. This analogy is elaborated in [60] where it is shown how AspectJ point-cut designators can be expressed as traversal strategies.

The LoDC has played a role in Adaptive Programming since its inception in 1992. Traversal code has always been considered as a separable element from the class graph and the visitors were written separately.

It is an empirical question how much traversal strategies (our concern-specific abstraction for the traversal concern) help in actual programming. Wu and Wand [61] have recently collected statistics on one system (DemeterJ [35]) where traversal strategies helped to create a system that is easier to evolve.

4. Strategies as Complex Request Interfaces

Implementing traversals is more complex than it first appears [37, 51, 52]. The commercially successful traversal technology that is widely used is XPath. It is used both for traversing XML documents and Java objects (JXPath).

Instead of XPath we use traversal strategies for reasons described in [37]. Traversal strategies and their implementations have gone through several iterations. We started with simple strategies of the form "from A bypassing X to B". Then we considered series-parallel strategies that used simple strategies and two operators: join and merge [52]. Our compilation algorithm for series-parallel strategies was limited in that it only worked on a limited set of class graph and traversal strategy combinations. We then generalized strategies along with their compilation algorithm [36, 37] making the algorithm both more general and faster. A technical report [38] shows a systematic derivation of the algorithm from a simple semantics that is easily understood by programmers. In a nutshell, for the traversal strategy "from x_1 to x_2 " follow edges in the set

$$POSS(x_1, x_2) = \{e \mid x_1 (\leq e \geq) (\leq C \geq)^* \leq x_2\}.$$

The formula can be informally summarized as: travel only through the paths where static types allow for the possibility of reaching a target class object. The details are in [38, 37].

Consider a variant of the salary summing problem for a company object, the original version is due to David Bock [18, 7]. The paperboy comes to your door and asks for a \$20 payment for the newspaper. You have some money in your wallet, but not enough. More money is in your kitchen cabinet but still not enough. For emergencies you have more money in your bedroom. You could let the paperboy know all those details but that would be inappropriate and could lead to misuse.

```
Instead of using (in class PaperBoy)
customer.wallet.money;
customer.apartment.kitchen.
    kitchenCabinet.money;
customer.apartment.bedroom.
    mattress.money;
```

we widen the interface of the Customer class and add a method `int customer.getPayment(...)`,

The method `getPayment` is implemented using a complex request in the form of a traversal strategy: "from Customer to Money" (similar to the "from Company to Salary" strategy). For the complete code, see [7].

A general problem of following the LoD is that it leads to wide class interfaces because there are so many ways of traversing through a complex object structure for different purposes. Therefore we use a declarative approach to specify where we want to go (e.g., "from Customer to Money"). Instead of sending many individual requests we send one complex request.

Such complex requests between a client and a server offer several advantages, proposed by Mitch Wand:

- Since the server gets a structured request, it has opportunities for optimization. For example, the traversal "from Customer to Money" does not have to visit the bathroom if the class graph says that there is no money in the bathroom.
- On the client side, knowledge about the request is expressed declaratively, rather than being embedded in the client's control structure, leading to improved locality of knowledge, modifiability, etc. In the paper boy example, we notice that information about the class graph is minimally duplicated in the complex request following the spirit of the LoD.
- In a distributed environment, complex requests minimize the number of messages that need to be sent between the client and the server.

Complex requests can also be aspectual. The `SummingVisitor` is an aspectual complex request to modify the execution of a traversal.

5. Other Law of Demeter Applications

The LoDC applies to other domains than writing methods.

5.1. Growth Plans for Class Graphs

The LoD also applies to constructing data structures (class graphs), not just constructing method bodies. Consider an apartment that is uninhabited: it consists of a kitchen, living room, bedroom and bathroom. When a tenant moves, the apartment gets enhanced with furnishings: beds, tables, etc. The bed is probably put into the bedroom but the furnishing information should be kept separate from the basic apartment structure [15].

In this application the modules are classes and they talk if they are in a has-a relationship. Each class is associated with a concern.

In the Demeter method we use the notion of a growth plan [27, 39] to incrementally develop an application. A

growth plan is a sequence of phases of increasingly larger class graphs where the final one is the class graph of the application. The sequence is constructed carefully so that each phase can be tested effectively.

Growth phases are good candidates to be encapsulated as aspects. But the extension points might have to be very fine-grained to achieve this. For example, in growth phase i we might have an assignment statement and in growth phase $i + 1$, the assignment statement will be wrapped by a conditional. This would require that we can select an assignment statement as an extension point. This would require a fine-grained model for expressing extension points that is currently not supported, e.g., by AspectJ.

5.2. Object Creation

We have hinted at how we can use adaptive methods to write shy code following the LoD and the DRY principle (Don't Repeat Yourself). Next we focus on object creation. We consider a constructor call to be a call to a class object. Our goal is to use a minimal number of such class objects in a given method because each one exposes information about its class, against the spirit of the LoD.

The traditional way to construct objects is to send messages to many class objects (or, equivalently, to have many constructor calls). It is better to parse an object description (a sentence in the language of a nonambiguous grammar): `Source.parse("object description")`. Consider the creation of an empty `FileSystem`-object. We could create explicitly a `FileSystem`-object, a `Directory`-object, a collection object for an empty list of files, a `DirectoryName`-object etc. Instead we could write an object description: "directory () root" saying that we want a directory called root containing no files. Files would be put between the parentheses.

Notice how the object description is more succinct than the long constructor calls that would duplicate a good chunk of the information in the class graph. The idea is to write a "schema" that defines both a set of classes and an LL(1) grammar and to use parser generation to create the parse function. This approach is used in the XML community by the Java data-binding tool (JAXB). But unfortunately, mark-up grammars (like XML schemas that are by their nature LL(1)) cannot provide the structure-shyness that is possible with grammars of LL(1) languages. An XML document is forced to repeat a lot of structure in the grammar (schema) while a sentence of an LL(1) grammar is not. Our tools Demeter/Flavors, Demeter/C++, DemeterJ, DAJ, and XAspects have used this technique successfully for many years.

The motto behind structure-shy object creation is: *Concrete syntax is more abstract than abstract syntax*. This is contrary to what is taught in a typical compiler class. Concrete syntax is more abstract because a sentence defines an

entire family of abstract syntax trees (objects) while an abstract syntax tree defines only one object. A particular element of the family is chosen by selecting a grammar. For example, "directory () root" represents a root directory object once we have chosen a particular schema for the file structure.

Richard Rasala has promoted the idea of creating *mature* objects versus the traditional way of creating *embryonic* objects that are incomplete [54]. With the Demeter data-binding technique, we practice creating mature objects in two steps: In the first step the parser creates a mature tree object and in the optional second phase, traversals create a completely linked, mature object ready for processing.

We have prepared a small collection of design patterns [28] that summarize the basic techniques of following the LoD: Class-Graph, Growth-Plan, Selective-Visitor, Structure-Shy-Object, Structure-Shy-Traversal. They are the foundation for the iterative development process of the Demeter Method [27].

6. Tool Support for the Law of Demeter

Several tools are available to follow the LoD.

6.1. LoD

AspectJ is an ideal language to write a LoD checker for Java programs [7, 34]. Our AspectJ checker does not deal with the LoDC, only with the LoD, although AspectJ has excellent capabilities to separate (crosscutting) concerns.

The LoD, when followed in an oo style, leads to scattering and many small methods. To address this scattering, we introduced traversal strategies and we developed the AP Library [9] that contains the key algorithms for processing traversal strategies. On top of the AP Library we have built DJ [49] and DAJ [31, 10] (besides the earlier DemeterJ system [35].) Those systems greatly facilitate traversal-related collaborations; DJ for Java and DAJ for AspectJ.

DJ takes the approach that the concepts behind traversal-related collaborations are made available as Java classes. The most important ones are ClassGraph, Strategy, Traversal and Visitor. DJ has run-time overhead and therefore we developed DAJ. DAJ uses the "declare" extension mechanism of AspectJ to declare strategies and traversals with visitors. This makes it very easy for AspectJ programmers, to follow LoD. DAJ avoids the run-time overhead of DJ by generating code statically. For further information on DJ and DAJ, see [31].

JXPath is another tool that makes it easier to follow the LoD. JXPath uses the XPath notation to navigate through Java objects. As described in [37], XPath is not (yet) as expressive as strategies with respect to structure-shyness.

6.2. LoDC

The AOSD community has developed a rich set of tools to support crosscutting concerns. See <http://www.aosd.net>. We started in 1996 with DemeterJ that supports the two concern-specific aspect languages covered in Crista Lopes thesis [41]. We developed a simple weaver that helped us to compile our concern-specific languages: class dictionaries, behavior files for traversal-related collaboration concerns, COOL files for the synchronization concern, and RIDL files for the data transfer concern. Recently, we have developed XAspects which improves DemeterJ by using AspectJ as the weaver and by providing a framework where many aspects can be integrated using plug-ins [56].

WebJinn [25] is a framework for concern separation in the Web application domain. WebJinn organizes code so that the presentation, functionality, control, and schema concerns are kept separate. In extant web application models, the schema concern is scattered: one cannot avoid referring to the schema from within the functionality and presentation. This violates the LoDC, because the schema concern contributes to neither of the other concerns. WebJinn follows the LoDC by decoupling the schema concern using a special referencing mechanism called schema extension points. This makes WebJinn a domain specific AOP framework for web applications.

7. Aspects and Components

Our goal is to encapsulate the information about a new concern to be added to a system as a component. A component is a black-box entity that can be deployed independently and that provides one or more specific services to the system [59]. The deployment is done through a connector in the component-connector style of software architecture [14]. Connectors provide a level of indirection that results in more reusable components. We would like to achieve the same reusability with aspects and therefore we proposed the concept of aspectual components [32] based on the adaptive plug-and-play component concept of [47]. An adaptive plug-and-play component consists of an "ideal" class graph (similar to a set of participants in contracts [16]) that represents the minimal structure to express the provided interface of the component, given the expected interface. Connectors are used to map the ideal class graph to the class graph where the component is used. Aspectual components are very similar: we have in addition to the regular methods also aspectual methods that have the power to modify other methods in an around-before-after style. Aspectual methods promote the notion of structured extension points: we do not just have a set of extension points where we want to perform the same extension action but we have a graph of

extension points and we usually have extension actions that differ from node to node. Components and aspects have a happy coexistence [46, 50, 58] but there are many open issues.

8. Open Problems

Issues around the LoD are pretty resolved but issues around how to follow the LoDC have many open questions.

- Controlling the power of aspects. Aspects, in the AspectJ style, are very powerful because they can reach inside existing code at many different places using low-level mechanisms. It is useful to study limited forms of aspects and aspects that use higher-level mechanisms to specify the coordinates of the enhancements.

The D*J tools (DemeterJ, DJ, DAJ) use a limited form of aspects (although the underlying algorithms are useful for aspects in general). The traversal-related collaboration aspects first extend the class diagram by introducing a set of node and edge traversing methods and then the aspects extend those methods when they traverse certain nodes and edges. By first introducing what is extended, aspects in Demeter are nicely localized as demonstrated by the weaving instruction in the form of a Java method call in Figure 1. The aspect `whenAndWhatToDo` is active during the traversal defined by aspect `whereToGo` which in turn operates in the context of `classGraph`. In the same program we might use multiple class graphs. Notice again the robustness with respect to changes of the class graph (structure-shyness): A significant change to the class-graph might not require any changes to the object description, `whereToGo` and `whenAndWhatToDo`.

This is one way of limiting the power of aspects and still gaining significant benefits from them. Other approaches are presented in [6, 33, 50]. We also need mechanisms so that code can protect itself from aspects without interfering too much with the power of aspects. One promising direction is to use contracts for aspects (and of course, the base code) [44]. The goal is to find a solution for the adaptation dilemma.

- Interaction between aspects. When multiple aspects influence the same extension point, we have issues of aspect composition and conflicts between aspects. Recent work in this area is [8].
- Reasoning about resource consumption. It is common practice to analyze the resource consumption of an algorithm but doing it for an aspect, such as an aspectual component, is much harder. There are two reasons for this. First, an aspect is parameterized over a program.

Doing a parameterized analysis over such complex parameters is hard. Second, the adaptation dilemma implies that an aspect is a shifting abstraction that adapts to actual parameters.

9. Conclusions

We have investigated the Law of Demeter (both LoD and LoDC) and its interesting ramifications. The LoD leads us to the dilemma of either violating it (writing code that is difficult to maintain) or to write code that crosscuts the class graph by being scattered across many classes and tangled with other concerns. To control the scattering and tangling, we used aspect-oriented techniques.

The LoD leads naturally to the LoDC. For example, the `whereToGo` concern should not be polluted with method calls about the `whenAndWhatToDo` concern. Or the base code should not be polluted with synchronization code or data transfer code.

We have hinted at a common structure, based on traversals, that underlies both aspect-oriented and adaptive programming. This connection will help to improve AOSD through better compilation algorithms and higher level extension languages.

Properly following the LoDC, or in other words, finding a proper decomposition into separable aspects, is still an issue with many questions attached (e.g., the adaptation dilemma). But the AOSD community continues to develop good solutions and it will ultimately succeed.

Acknowledgments I would like to thank my current research group (Doug Orleans, Johan Ovlinger, Jeffrey Palm, Therapon Skotiniotis, Pengcheng Wu, Sergei Kojarski) and my CCIS colleagues: Will Clinger, Matthias Felleisen, David Lorenz, Richard Rasala and Mitchell Wand for their inspiration and support. Many thanks for the feedback from Jonathan Aldrich, Ian Holland, Gregor Kiczales, Crista Lopes, Joe Loyall, Mira Mezini, Macneil Shonle, André van der Hoek and Wim Vanderperren. This work was supported most recently by NSF grant CCR 0098643 and DARPA grant F33615-00-C-1694 (under subcontract from BBN), a grant from ABB Switzerland, an Eclipse Fellowship from IBM and a grant from Neeraj Sangal.

References

- [1] *Companion of the 18th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, Anaheim, California, 2003. ACM Press.
- [2] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa. Abstracting Object Interactions Using Composition Filters. In R. Guerraoui, O. Nierstrasz, and M. Riveill, editors, *Proceedings of the ECOOP'93 Work-*

- shop on Object-Based Distributed Programming*, volume 791, pages 152–184. Springer-Verlag, 1994.
- [3] J. Aldrich. Open Modules: A Proposal for Modular Reasoning in Aspect-Oriented Programming. In *Proceedings of the FOAL workshop of the 3rd international conference on Aspect-Oriented Software Development*, 2004.
- [4] D. Batory and B. Geraci. Composition validation and subjectivity in genova generators. *IEEE Transactions on Software Engineering*, 23(2):67–82, February 1997.
- [5] A. Colyer, A. Rashid, and G. Blair. On the separation of concerns in program families. Technical Report COMP-001-2004, Computing Department, Lancaster University, UK, 2004.
- [6] D. Dantas and D. Walker. Aspects, information hiding and modularity. Technical report, Princeton University, November 2003.
- [7] Demeter Research Group. Law of Demeter Home Page. <http://www.ccs.neu.edu/home/lieber/LoD.html>.
- [8] R. Douence, P. Fradet, and M. Südholt. Composition, Reuse and Interaction Analysis of Stateful Aspects. In K. Lieberherr, editor, *Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 141–150. ACM Press, 2004.
- [9] Doug Orleans and Karl J. Lieberherr. AP Library: The Core Algorithms of AP: Home page. <http://www.ccs.neu.edu/research/demeter/AP-Library/>.
- [10] Doug Orleans and Karl J. Lieberherr. DAJ: Demeter in AspectJ home page. <http://www.ccs.neu.edu/research/demeter/DAJ/>.
- [11] G. Duzan, J. Loyall, R. Schantz, R. Shapiro, and J. Zinky. Building Adaptive Distributed Applications with Middleware and Aspects. In K. Lieberherr, editor, *Proceedings of the 3rd international conference on Aspect-Oriented Software Development*, pages 66–73. ACM Press, 2004.
- [12] T. Elrad, R. Filman, and A. Bader. Aspect-Oriented Programming. *Communications of the ACM*, 44(10):28–97, 2001.
- [13] R. E. Filman and D. P. Friedman. Aspect-Oriented Programming is Quantification and Obliviousness. In *Workshop on Advanced Separation of Concerns, OOPSLA*, Minneapolis, USA, 2000. <http://ic-www.arc.nasa.gov/ic/darwin/oif/leo/filman/text/oif/aop-is.pdf>.
- [14] D. Garlan and M. Shaw. An introduction to software architecture. In *Advances in Software Engineering and Knowledge Engineering*, volume I. World Scientific Publishing Company, 1993.
- [15] W. Harrison and H. Ossher. Subject-oriented programming (A critique of pure objects). In *Proceedings OOPSLA '93*, pages 411–428, Oct. 1993. ACM SIGPLAN Notices, volume 28, number 10.
- [16] I. M. Holland. *The Design and Representation of Object-Oriented Components*. PhD thesis, Northeastern University, 1993.
- [17] A. Hunt and D. Thomas. *The Pragmatic Programmer*. Addison-Wesley, 2000.
- [18] A. Hunt and D. Thomas. The art of enbugging. *IEEE Software*, pages 10–11, January/February 2003.
- [19] W. L. Hürsch and C. V. Lopes. Separation of concerns. Technical Report NU-CCS-95-03, College of Computer Science, Northeastern University, Boston, MA, February 1995.
- [20] IBM Research Team. Hyper/J home page. <http://www.research.ibm.com/hyperspace/HyperJ/HyperJ.htm>.
- [21] M. A. Jackson. *Software Requirements and Specification*. ACM Press, 1995.
- [22] I. Jacobson. Language support for changeable large real time systems. In *Conference proceedings on object-oriented programming systems, languages and applications*, pages 377–384. ACM Press, 1986.
- [23] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An Overview of AspectJ. In J. Knudsen, editor, *European Conference on Object-Oriented Programming*, Budapest, 2001. Springer Verlag.
- [24] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming*, pages 220–242. Springer Verlag, 1997.
- [25] S. Kojarski and D. H. Lorenz. Domain driven web development with WebJinn. In *Companion of the 18th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications* [1], pages 53–65. Special Track on Domain-Driven Development.
- [26] K. J. Lieberherr. Component enhancement: An adaptive reusability mechanism for groups of collaborating classes. In J. van Leeuwen, editor, *Information Processing '92, 12th World Computer Congress*, pages 179–185, Madrid, Spain, 1992. Elsevier.
- [27] K. J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, 1996. 616 pages, web book at www.ccs.neu.edu/research/demeter.
- [28] K. J. Lieberherr. Pattern Language for Adaptive Programming. <http://www.ccs.neu.edu/research/demeter/adaptive-patterns/pattern-lang-conv>, Feb. 1997.
- [29] K. J. Lieberherr and I. Holland. Assuring good style for object-oriented programs. *IEEE Software*, pages 38–48, September 1989.
- [30] K. J. Lieberherr, I. Holland, and A. J. Riel. Object-oriented programming: An objective sense of style. In *Object-Oriented Programming Systems, Languages and Applications Conference*, in *Special Issue of SIGPLAN Notices*, number 11, pages 323–334, San Diego, CA, September 1988. A short version of this paper appears in *IEEE Computer Magazine*, June 1988, Open Channel section, pages 78–79.
- [31] K. J. Lieberherr and D. Lorenz. Coupling Aspect-Oriented and Adaptive Programming. In R. Filman, T. Elrad, S. Clarke, and M. Aksit, editors, *Aspect-Oriented Software Development*. Addison-Wesley, 2004. In press.
- [32] K. J. Lieberherr, D. Lorenz, and M. Mezini. Programming with Aspectual Components. Technical Report NU-CCS-99-01, College of Computer Science, Northeastern University, Boston, MA, March 1999.

- [33] K. J. Lieberherr, D. Lorenz, and J. Ovlinger. Aspectual collaborations – combining modules and aspects. *The Computer Journal*, 46(5):542–565, September 2003. <http://www.ccs.neu.edu/research/demeter/papers/ac-aspectj-hyperj>.
- [34] K. J. Lieberherr, D. H. Lorenz, and P. Wu. A Case for Statically Executable Advice: Checking the Law of Demeter With AspectJ. In M. Aksit, editor, *Second International Conference on Aspect-Oriented Software Development*, pages 40–49, Boston MA, 2003. ACM Press.
- [35] K. J. Lieberherr and D. Orleans. Preventive program maintenance in Demeter/Java (research demonstration). In *International Conference on Software Engineering*, pages 604–605, Boston, MA, 1997. ACM Press.
- [36] K. J. Lieberherr and B. Patt-Shamir. Traversals of Object Structures: Specification and Efficient Implementation. Technical Report NU-CCS-97-15, College of Computer Science, Northeastern University, Boston, MA, Sep. 1997. <http://www.ccs.neu.edu/research/demeter/AP-Library/>.
- [37] K. J. Lieberherr, B. Patt-Shamir, and D. Orleans. Traversals of Object Structures: Specification and Efficient Implementation. *ACM Transactions on Programming Languages and Systems*, March 2004.
- [38] K. J. Lieberherr and M. Wand. Navigating through object graphs using local meta-information. Technical Report NU-CCS-2001-05, Northeastern University, May 2001. <http://www.ccs.neu.edu/research/demeter/biblio/new-strategy-antics.html>.
- [39] K. J. Lieberherr and C. Xiao. Object-Oriented Software Evolution. *IEEE Transactions on Software Engineering*, 19(4):313–343, April 1993.
- [40] C. Lopes. AOP: A Historical Perspective. In R. Filman, T. Elrad, S. Clarke, and M. Aksit, editors, *Aspect-Oriented Software Development*. Addison Wesley, 2004. In press.
- [41] C. I. V. Lopes. *D: A Language Framework for Distributed Programming*. PhD thesis, Northeastern University, 1997. 274 pages.
- [42] C. V. Lopes, P. Dourish, D. H. Lorenz, and K. J. Lieberherr. Beyond AOP: Toward Naturalistic Programming. In *Companion of the 18th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications* [1], pages 34–43. Onward! Track.
- [43] C. V. Lopes and K. J. Lieberherr. Abstracting process-to-function relations in concurrent object-oriented applications. In R. Pareschi and M. Tokoro, editors, *European Conference on Object-Oriented Programming*, pages 81–99, Bologna, Italy, 1994. Springer Verlag, Lecture Notes in Computer Science.
- [44] D. H. Lorenz and T. Skotiniotis. Contracts and aspects. Technical Report NU-CCIS-03-13, College of Computer and Information Science, Northeastern University, Boston, MA 02115, Dec. 2003.
- [45] 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*, Enschede, Netherlands, 2002.
- [46] M. Mezini and K. Ostermann. Conquering Aspects with Caesar. In M. Aksit, editor, *Second International Conference on Aspect-Oriented Software Development*, pages 90–100, Boston MA, 2003. ACM Press.
- [47] M. Mezini and K. J. Lieberherr. Adaptive plug-and-play components for evolutionary software development. In C. Chambers, editor, *Object-Oriented Programming Systems, Languages and Applications Conference*, in *Special Issue of SIGPLAN Notices*, number 10, pages 97–116, Vancouver, October 1998. ACM.
- [48] D. Orleans. Incremental programming with extensible decisions. In G. Kiczales, editor, *First International Conference on Aspect-Oriented Software Development*, Enschede, The Netherlands, 2002. ACM Press.
- [49] D. Orleans and K. J. Lieberherr. DJ: Dynamic Adaptive Programming in Java. In *Reflection 2001: Meta-level Architectures and Separation of Crosscutting Concerns*, Kyoto, Japan, September 2001. Springer Verlag. 8 pages.
- [50] J. Ovlinger. *Combining Aspects and Modules*. PhD thesis, Northeastern University, 2004. Draft version.
- [51] J. Palsberg, B. Patt-Shamir, and K. J. Lieberherr. A new approach to compiling adaptive programs. *Science of Computer Programming*, 29(3):303–326, 1997.
- [52] J. Palsberg, C. Xiao, and K. J. Lieberherr. Efficient implementation of adaptive software. *ACM Transactions on Programming Languages and Systems*, 17(2):264–292, Mar. 1995.
- [53] G. Polya. *How to solve it*. Princeton University Press, 1949.
- [54] R. Rasala. Embryonic object versus mature object: object-oriented style and pedagogical theme. *SIGCSE Bulletin*, 35(3):89–93, 2003.
- [55] L. M. Seiter, J. Palsberg, and K. J. Lieberherr. Evolution of Object Behavior using Context Relations. *IEEE Transactions on Software Engineering*, 24(1):79–92, January 1998.
- [56] M. Shonle, K. J. Lieberherr, and A. Shah. XAspects: An Extensible System for Domain Specific Aspect Languages. In *Companion of the 18th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications* [1], pages 28–37. Special Track on Domain-Driven Development.
- [57] J. Sung. Aspectual Concepts. Technical Report NU-CCS-02-06, Northeastern University, June 2002. Master’s Thesis, <http://www.ccs.neu.edu/home/lieber/theses-index.html>.
- [58] D. Suvée, W. Vanderperren, and V. Jonckers. JAsCo: an aspect-oriented approach tailored for component based software development. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 21–29. ACM Press, 2003.
- [59] C. Szyperski. *Component Software*. Addison-Wesley, 1999.
- [60] P. Wu and K. J. Lieberherr. Compilation of Pointcut Designators using Traversals. Technical Report NU-CCIS-03-16, Northeastern University, December 2003.
- [61] P. Wu and M. Wand. An Empirical Study of the Demeter System. In *Proceedings of the SPLAT workshop of the 3rd international conference on Aspect-Oriented Software Development*, 2004.