

Adaptive Software Design

Designing adaptive software using the Demeter™ method

John P. Daigle, Arash Akhlaghi

Department of Computer Science
Georgia State University

04.16.07

Outline

- 1 Introduction
 - Problem
 - Definitions
- 2 The Law of Demeter
 - Introduction
 - Notation
- 3 Case Study
 - Description
 - Weak Law
 - Strong Law
- 4 Conclusion

Outline

- 1 Introduction
 - Problem
 - Definitions
- 2 The Law of Demeter
 - Introduction
 - Notation
- 3 Case Study
 - Description
 - Weak Law
 - Strong Law
- 4 Conclusion

- Computers are
 - Fast
 - Consistent
 - Predictable
- Computers are not
 - Smart
 - Flexible
 - Adaptable

Need Flexibility

As computers are deployed in a wider range of applications, we need software that is reusable, adaptable, smart, and flexible.

Adaptability

Adaptive Software

Adaptive Software software is *portable*. The developer can easily alter the code to suit a number of environments.

Adaptable Software

Adaptable Software is *flexible*. The software can either alter itself, or it its behavior, to suit a number of environments.

Object Orientation

Principles

Object oriented programming solve the problem of adaptable and adaptive software through two basic principles.

- Encapsulation
- Decoupling

Encapsulation

Encapsulation

The goal of encapsulation is to create a **black box** that returns values by an opaque internal mechanism.

- hides local changes
 - promotes modularity
-
- Information Hiding
 - Information Restriction
 - Localization of Information

Decoupling

Freedom!

The principle of decoupling is the reduction of functional dependencies between classes. A class that is tightly coupled to another calls the methods of that class, a class that is loosely coupled, or decoupled, does not.

- Narrow Interface
- Coupling control

Questions

- How do we know we have written good OO programs?
- Is there a metric for correct object oriented style?

Outline

- 1 Introduction
 - Problem
 - Definitions
- 2 The Law of Demeter
 - Introduction
 - Notation
- 3 Case Study
 - Description
 - Weak Law
 - Strong Law
- 4 Conclusion

The Law of Demeter is a methodology for solving the problem of adaptive and adaptable software:

- Reduces the burden on programmers during development, LoD easier to build
- Reduces the maintenance cost, LoD programs are easier to understand

Law of Demeter

For all classes \mathcal{C} , and for all methods \mathcal{M} attached to \mathcal{C} , all objects to which \mathcal{M} sends a message must be instances of classes associated with the following classes:

- 1 The argument classes of \mathcal{M} (including \mathcal{C})
- 2 The instance variable classes of \mathcal{C} [1]

Addressing the Goals of OOP

Coupling Control The Law of Demeter reduces range of method calls

Information Hiding Structure hiding, methods cannot see into the object hierarchy

Information Restriction Focus on localizing type information

Narrow interfaces Promotes naivety of other classes

Variations

Strong LoD

The strong law of Demeter defines instance variables as being exclusively the instance variables of a given class. Inherited instance variable types may not be passed messages.

Weak LoD

The Weak Law defines instance variables as being both the instance variables with make up a given class and any instance variables inherited from other classes.

Production Rules

Constructive Production

- Builds a class from other classes

$$C = \langle id_1 \rangle SC_1 \dots \langle id_n \rangle SC_n \quad (1)$$

- Each Class is made up of variables that have an id, id_i and a type SC_i
- For any instance of C , the identifier id_i refers to a member of class SC_i

Computer =

<keyboard> Input_Device

<monitor> Output_Device

<mouse> Input_Device

Other Productions

Alternation Production

- Expression of a union type

$$C : A|B \quad (2)$$

- C is either a member of class A or class B , exclusively.

```
USB-Device : Input_Device | Output_Device
```

Repetition Production

- A list or array construction

$$C \{A\} \quad (3)$$

- Members of C are lists of zero or more instances of A

Inheritance and Signature

Inheritance

- Multiple inheritance is not strictly disallowed

```
Tablet-Screen =  
    <screen> Tablet_Device  
    *inherit* Input_Device.
```

- This is inheritance, not composition, *Tablet_Screen is an Input_Device.*

Signature

The set formed by taking the union of the set of methods attached to a class \mathcal{C} and the set of methods attached to super classes of \mathcal{C} .

Weak and Strong Law

Strong Law of Demeter

Instance variables are only the instance variables of the class.
Inherited instance variable types may not be passed messages.

Weak Law of Demeter

Instance variables may be inherited.

Outline

- 1 Introduction
 - Problem
 - Definitions
- 2 The Law of Demeter
 - Introduction
 - Notation
- 3 Case Study
 - Description
 - Weak Law
 - Strong Law
- 4 Conclusion

Case Description

How do we calculate the weight of a collection (a “Basket”) of fruit of different types?

```
FruitBasket = Basket Fruits.  
Fruits ~ {Fruit}.  
Fruit : Apple | Orange | Plum  
        *common* <weight> Number.  
Basket = . Apple = . Orange = . Plum = .
```

- Fruit is considered a *Union* of Apple, Orange and Plum
- Apple, Orange, and Plum *inherit* from Fruit

Method

We now want to write a method which will compute the prepared weight of a basket of fruit.

Assumptions

- The prepared weight of a piece of fruit is the weight of the fruit less skin, core and/or pit.
- Each fruit has a different formula for computing the prepared weight from the gross weight.
- Prepared weights of apples, oranges and plums are 85, 80 and 65 percent of gross weight (respectively).

Code

```
(defmethod (FruitBasket :compute-weight) ()  
  (send Fruits:compute-weight))
```

```
(defmethod (Fruits :compute-weight) ()  
  (loop for each-fruit in child sum  
        (send each-fruit :compute-weight)))
```

```
(defmethod (Apple :compute-weight) ()  
  (* weight 0.85))
```

```
(defmethod (Orange :compute-weight) ()  
  (* weight 0.8))
```

```
(defmethod (Plum :compute-weight) ()  
  (* weight 0.65))
```

Analysis

```
(defmethod (Apple :compute-weight) ()  
  (* weight 0.85))
```

The use of the inherited instance variable `weight` within the `:compute-weight` methods attached to `Apple`, `Orange` and `Plum` is in violation of the strong interpretation of the law.

code

```
(defmethod (FruitBasket :compute-weight) ()
  (send Fruits:compute-weight))

(defmethod (Fruits :compute-weight) ()
  (loop for each-fruit in child sum
        (send each-fruit :compute-weight)))

(defmethod (Apple :compute-weight) ()
  (send self :get-percent-weight 0.85))

(defmethod (Orange :compute-weight) ()
  (send self :get-percent-weight 0.8))

(defmethod (Plum :compute-weight) ()
  (send self :get-percent-weight 0.65))

(defmethod (Fruit :get-percent-weight)
  (percent)
  (* weight percent))
```

Analysis

The strong solution requires an extra method call. Is there an advantage to the strong solution?

Adaptable Software

The problem that we meant to solve was to create software that is adaptable. Will our weak solution work on the moon?

```
(defmethod (Orange :compute-weight) ()  
  (* weight 0.8))
```

If we move the fruit basket to the moon, we will have to change the value of the `weight` variable, because `weight` is a function of gravity! So our code is not portable to the moon.

Using the Strong Solution

We now know that we need a solution that calculates weight based on mass and gravity.

```
FruitBasket = Basket Fruits.  
Fruits ~ {Fruit}.  
Fruit : Apple | Orange | Plum  
    *common* <weight> PlanetWeight.  
PlanetWeight =  
    <mas> Number <gravity> Number.
```

The only modified class in this class dictionary is Fruit.

Alter the Code

We modify the `get-percent-weight` method

```
(defmethod (Fruit :get-percent-weight)
  (percent)
  (send weight :get-percent-weight percent) )

(defmethod (PlanetWeight :get-percent-weight)
  (percent)
  (* (* mass gravity) percent))
```

Strong Interpretation

The Strong law provides

- Decoupling
- Encapsulation
- Adaptability
- Scalability

Outline

- 1 Introduction
 - Problem
 - Definitions
- 2 The Law of Demeter
 - Introduction
 - Notation
- 3 Case Study
 - Description
 - Weak Law
 - Strong Law
- 4 Conclusion

This was a historical view:

- Law of Demeter never overly popular
- Overtaken by design patterns
- Considered obvious by modern standards of OO design

However, the LoD is not defunct

- Still a valuable and simple principle
- Easier to remember than a detailed list of principles
- Scales to Aspect Oriented Programming
- Can be checked by a compiler

For Further Reading I



K. Lieberherr, I. Holland, and A. Riel.

Object-oriented programming: an objective sense of style.
In *OOPSLA '88: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 323–334, New York, NY, USA, 1988. ACM Press.



K. J. Lieberherr.

Demeter/adaptive programming.
SIGSOFT Softw. Eng. Notes, 25(1):100–101, 2000.



K. J. Lieberherr.

Controlling the complexity of software designs.
In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 2–11, Washington, DC, USA, 2004. IEEE Computer Society.

For Further Reading II



K. J. Lieberherr and D. Orleans.

Preventive program maintenance in demeter/java.

In ICSE '97: Proceedings of the 19th international conference on Software engineering, pages 604–605, New York, NY, USA, 1997. ACM Press.



K. J. Lieberherr and A. J. Riel.

Demeter: a case study of software growth through parameterized classes.

In ICSE '88: Proceedings of the 10th international conference on Software engineering, pages 254–264, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.



K. J. Lienberherr.

Formulations and benefits of the law of demeter.

SIGPLAN Not., 24(3):67–78, 1989.

