LoDChecker Requirements

October 3, 2002

Contents

1 The Law of Demeter

This document details requirements for a piece of software called LoDChecker. The program will analyze another program and report violations of the Law of Demeter. The first section of this document will describe the Law of Demeter in some detail, and the following section will enumerate and describe the functional and non-functional requirements for LoDChecker.

1.1 What Is It?

The Law of Demeter is a style rule for object-oriented programming that aims to minimize the coupling between classes in the system. Software developers who follow this law will find that their program is much easier to extend and maintain, and each unit of code may be comprehended independently of the rest of the system. [LHR88] The LoD encourages the developer to write "shy" code by adhering to the principles of encapsulation and low coupling.[HT00]

1.2 Forms of the Law of Demeter

1.2.1 Class Form

The class form of the LoD, though less stringent, has the advantage of being statically checkable on any object-oriented program. The lack of stringency stems from the fact that the programmer is free to violate the spirit of this law while adhering to its letter. This form of the law stipulates that any method m of class C may only call methods belonging to:

- class C
- the classes of m's arguments
- immediate part classes of class C:
  - classes that are return types of methods of class C
  - classes of data members of class C
  - classes of objects constructed within the scope of m

[LH89]

This formulation implicitly includes static methods since they each belong to a particular class.

1.2.2 Object Form

The object form of the LoD is not statically checkable; checking must be done dynamically. Its stipulations are stricter than those of the class form, however, so this dynamic program analysis offers greater benefit than the class form. This version states that, within some method m, messages may only be sent to objects belonging to the following categories:

- Objects that are parameters of m (implicitly includes enclosing object).
- An immediate part of the message sender.
  - An object returned by a message sent to the enclosing object.
  - Data members of the enclosing object.
  - Collection elements, if the collection is a part of the enclosing object.
- An object that is constructed within m.
- A global object.

[LH89]

Handling of static methods is unspecified by the object form, however, we will define the behavior of LoDChecker with respect to static method checking in the functional requirements section.

1.2.3 Strong vs. Weak

The strong form is a constraint that may be applied to either the class or the object form. In the context of the class form, the constraint indicates that the immediate part classes of a given class C do *not* include the classes of inherited members. In the object form of the LoD, data members that receive messages must be defined within C; inherited members are *not* valid message targets.

The weak form does not impose this constraint. In class form, messages may be sent to both inherited members and their classes, and, in object form, messages may be sent to inherited objects.[LH89]

2 Software Requirements

This section of the document will enumerate and describe the key requirements of LoDChecker. User interface concerns are not addressed at this point.

2.1 Non-functional Requirements

2.1.1 Extensibility

LoDChecker must be written in a modular and extensible style such as aspect-oriented. In particular, the software will mature and become part of a code refactoring tool. This refactoring tool will be a plug-in to the Eclipse IDE (http://www.eclipse.org).

2.1.2 Works with Java

LoDChecker will be applied to programs written in Java. LoDChecker, therefore, must be able to run on any system that is capable of running Java programs. LoDChecker will be dynamic; it therefore may assume that the Java program to be analyzed is syntactically correct / compilable.

2.2 Functional Requirements

2.2.1 Check various LoD forms.

LoDChecker must be able to dynamically check the different forms of the LoD. The tool must be able to check both the class form and the object form, and it must also be able to check on and differentiate between strong form violations and weak form violations. Static analysis is not required.

2.2.2 Checking Modes

LoDChecker must allow the user to choose the form(s) s/he wishes to check. The user must be able to select either class form or object form. Selecting both is redundant since the object form encompasses the restrictions of the class form. The user must also be able to select either weak form or strong form. The allowable permutations, then, are:

- Class / Weak

- Class / Strong
- Object / Weak
- Object / Strong

2.2.3 Handling of static methods

Static methods fall naturally into the requirements set forth by the class form of the LoD. This issue becomes ambiguous in the object form because static methods belong to classes rather than objects. If LoDChecker is analyzing a program for object form violations, it must also report violations due to calls of static methods, per the class form stipulations.

2.2.4 Report LoD violations.

When LoDChecker detects a violation, the tool must produce a useful report of this violation to the user. A useful report must consist of the following information (at minimum):

- the method and class in which the violation occurs.
- the offending method call's name and the class where it is defined.
- which form(s) of the Law of Demeter are violated (class, object, strong, and/or weak).

The file name and source code line number of the violation are suggested but not mandatory.

3 Use Cases

3.1 Use Case: Check Object Weak LoD

3.1.1 Characteristic Information

Goal : User wants to check a Java program for compliance with the Object / Weak LoD.

Scope : Whole program.

Level : Summary

Preconditions : Target Java program compiles and can be executed.

Success end condition : Java developer has a report of LoD violations.

Failed end condition : Tool failed to analyze program.

Primary actor : Java program developer.

Trigger : Developer activates LoD on target Java program.

3.1.2 Main Success Scenario

1. Developer selects Object Form and Weak Form in LoDChecker.
2. Developer specifies the target Java program to be analyzed.
3. Developer runs LoDChecker.
4. LoDChecker produces a report of detected LoD violations.
5. Developer reviews report and corrects violations.

3.1.3 Extensions

4a. LoDChecker finds no violations: Report indicates no violations are found.

5a. Developer wishes to keep a violation as is: Developer ignores that violation or reruns LoDChecker with more relaxed constraints.

3.1.4 Related Information

Priority : Top.

Performance : Checker should run in polynomial time.

3.1.5 Schedule

Due date : Release 1.0

3.1.6 Open Issues

- What happens if user's program crashes during execution?
- What if user's program is time critical and LoDChecker introduces too much overhead?
- User interface? Provide as Java package or external tool?

A Glossary

Coupling A measurement of the degree with which system components depend directly on other system components. Low coupling generally leads to easier system extension and maintenance.

Dynamic checking LoDChecker will analyze the program as it is running rather than analyzing the source code before or during compilation.

Enclosing object The 'this' or 'self' in most object-oriented languages.

Law of Demeter Read section one for a description.

LoD Abbreviation for "Law of Demeter".

LoDChecker The name of our software tool; it checks an input program's structure based on the Law of Demeter and reports violations.

Preferred supplier Any of the set of objects and classes that may legally (wrt. LoD) receive messages from a given method.[LH89]

"Shy" code Code that is both encapsulated and does not make assumptions about the rest of the system.[HT00]

Violation A method call that does not adhere to the stipulations prescribed by the Law of Demeter.

References

[HT00] Andrew Hunt and David Thomas. *The Pragmatic Programmer*. Addison-Wesley, Boston, 2000. ISBN 0-201-61622-X.

[LH89] Karl J. Lieberherr and Ian Holland. Assuring good style for object-oriented programs. *IEEE Software*, pages 38–48, September 1989.

[LHR88] Karl J. Lieberherr, Ian Holland, and Arthur 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.