Requirement Document: Designing a Law of Demeter style rule checker.

Introduction:

For this project we wish to develop a source code style checker. Specifically, we wish to design a tool that checks code to see that it conforms to the Law of Demeter (LoD). Often, in large software projects, code can become to complex, to closely coupled, with information passing that is simply uncontrollable. These problems, along with others, lead to code that is un-maintainable, un-adaptable and generally not practical for long term use and development. The LoD, named after the Demeter system, was created to avoid these problems through the use of a high level class structure. LOD primarily achieves this goal by requiring that code be "shy". Information is hidden, and objects have a simple, one to one, relationship with one another. No object, method, or function will traverse three or four levels of objects to reach a variable. Objects will make a request to another function or object with which it has a direct relationship, and that function or object will return an answer. There are also numerous interpretations of this law, so this program must have enough flexibility to support them all.

Glossary:

Law of Demeter (LoD): the fundamental law that our style checker will follow. The basic assertion of this law, for the object variation (our base case), is that we should "minimize coupling between modules."

Module Coupling: Having object modules that have direct dependencies with one another. These dependencies cause the developer to concern himself/herself with code outside of the immediate scope of the code with which they're dealing.

Portability: The ability to move one piece of code, or one application or module, from one piece of code to another. Making code portable, or "adaptable", is the main focus of this tool.

Object Form (OF-LoD): A version of the LoD that requires methods only make calls to "preferred" suppliers within the object.

Class Form (CF-Lod): Version of the LoD that states that methods inside a class may only make calls the methods attached to that class. Especially useful with static code.

Strong Form: LoD variation that declares class instances to include only variables comprising that class.

Weak Form: Similar to Strong form of the LoD but also allows the use of inherited variables.

Instance Variable: Variable that is an instance of the class that it is in.

Inheritance: When one object is a "child" of another object in the sense that it has all of its "parents" information in addition to the objects and variables it has defined itself.

Argument to a method: A variable that has been passed in to a method.

Global Variable: variable that can be seen by (whose scope is) the entire program.

Functional Requirements:

To support this style checking tool we must support the Law of Demeter. However, we cannot generalize this law, we must be specific in what the law allows and in that same sense we must support the four main variations (i.e. Object, Class, Strong, Weak). These variations should be supported as follows:

1) To support the object form of the Law of Demeter the calling object must be either
    a. an instance variable
    b. an argument that has been based in
    c. an object created in the method
    d. a global variable
2) To support the Class form of the LoD the class in the method must be either
    a. a static instance variable
    b. a static or new object
    c. an argument to the method
    d. the return type of the method.
3) To support the strong form of LoD the program must ensure that instance variables to the object are only the instance variables that make up the given class.
4) Support for the weak form of the LoD is similar to the Strong form, but it also allows variables that have been inherited to be included in the group of instance variables.

The result of implementing this law will be a reduction of a classes "response set", thus simplifying the source codes complexity and reducing the probability for error. There should be less coupling of objects that shall lead to easier unit testing. Each form has specific benefits and trade-offs that the developer should be aware of. Object form gives control of module coupling but requires many methods that do little work. Class form provides simple definitions with static programming but is not especially useful with dynamic objects. Strong form guarantees that any change to data structure will only affect the methods using this structure, but may require special methods for special jobs, thus complicating the code. The Weak form provides some of the advantages of the strong form, mainly that program maintenance is improved, but that changes to underlying data structures require the modification of modules that use those structures as well as any inherited class.

Use Case:

A. Characteristic Information
    - Goal in context: Developer has to write a program that will both ensure the use of LoD coding standards, as well as correct code that fails the requirement.
    - Scope: Course.
    - Level: Summary.
    - Preconditions: Some code hierarchy already in place, existing code may or may not conform to LoD standards.

- Success end condition: All code is confirmed to be up to LoD development standards and consequently be deemed sound OO source code.
- Failed end condition: Resulting source code is not LoD compliant.
- Primary actor: Developer.
- Trigger: Professor requests new program works on existing source code.

B. Main Success Scenario
    a. Program written is knowledgeable of all variations of LoD and is able to correctly identify and fix a discrepancy.
    b. All modules are loosely coupled, with each one only talking to their "preferred" source.
    c. Source code comes together and produces functional software.
    d. Professor/grader receives fully functional version of requested software
    e. Source code is in a fully LoD state.

C. Extensions
    a. Correct any Java based program in the future (to ensure LoD rules are upheld).

D. Variations
    a. Code may be compliant with any of the variations of LoD.

E. Related Information
- Priority: Top
- Performance target: Program that completely and correctly checks source code to ensure it is up to the defined standards.
- Superordinate use case: Source code complies and programs still run after the check/conversion to LoD.
- Subordinate use case: program must be in LoD form itself.
- Channel to primary actor: phone or email
- Secondary actors: none.

F. Schedule
- Due end of quarter

G. Open issues
- How deeply imbedded are the method calls going to be that our program is going to have to work with?

Requirements Specification:

This program must support the development and evolution of a strong class hierarchy that is both adaptable and expandable. Source code resulting from this program must be flexible and behave in a well defined manner. Enforcing this law implies that our style checker will ensure the reduction in code coupling, encourage information hiding, restrict information to only the methods who should see it, reduce the

number of interfaces, restrict the size of the interface and explicitly state what can be used by the methods. In general, this tool needs to limit the number of dependencies in the code.

The basic form for LoD that we will support (object form) restricts the use of objects and methods unless the method is an instance variable, an argument, a locally created object or a global variable to the requesting object. These rules are not open for interpretation as they form the basis of our style checker. This implies the program must ensure that no object goes through another object to reach a third object. One object talks to another object that it owns, inherits from or has been passed. That object can, in turn, ask a third object and return the answer, but it must not be used as a communication link. This communication is said to occur between "preferred" objects and the code is said to be written in a "shy" manner. The coupling of modules must be tightly controlled to keep the resulting source code from spiraling out of control to the point where it loses its flexibility, becomes un-maintainable and very costly to the user and the developer.

The four rules above are laid out specifically to control module coupling. Beyond loose module coupling exists a number of other benefits that the Demeter system provides and that the program must ensure. Information hiding should be enforced. Once the coupling is under control there is no reason for outside objects to be aware of a given objects private members. The outside object only needs a set of methods that provide access to the information that is useful and appropriate. Beyond information hiding is information restriction. No method should be written to provide information that is not necessary. We must also be concerned with the interface structure. First, we want to limit the number of interfaces and restrict the number of classes that can be used by a method. Second, while limiting the number of the interfaces, we should also limit the size and, consequently, the amount of data that is passed around by the methods. Last, we want to be explicit in defining which classes can be used by which interface.