Some Object-Oriented Design Principles

Open/Closed Principle

(Robert C. Martin)

- Classes should be both “open” and “closed”:
  - Open: means that the class can be extended through inheritance.
  - Closed: means that the functionality of a class, once set, should not be modified retroactively.

- In other words, add functionality by adding new code, not rewriting old code.

Liskov Substitution Principle (LSP)

As popularly stated:

A member of a derived class must also make sense when used as a member of the base class.

For example, if a method has an object of a class as an argument, the same method should be able to work with an object of a derived class.

As originally stated:

Let $\varphi(x)$ be a property provable about objects x of type T.
Then $\varphi(y)$ should be true for objects y of type S, where S is a subtype of T.

(Prof. Barbara Liskov, M.I.T.)

B. Liskov and J. Wing, A behavioral notion of subtyping, ACM TOPLAS, **16**, 6 (Nov. 1994), 1811-1841.

LSP **Corollary**
(Robert C. Martin)

- Functions that use *pointers* or *references* to base classes must be able to use objects of derived classes **without differentiation** as to base vs. derived.

Impact of LSP

- We should not have to make special-cases of behaviors of derived methods.

  This would be an indication that they are not behaving like members of the parent class; maybe rethink this membership.

Thought Questions

- Suppose we have a class RealValuedFunction and a sub-class DifferentiableFunction. Where should the method getDerivative() be placed?
- Should a class Bird have a method FlyDistance()?
- Suppose we have a class Year.
  Should we have a subclass LeapYear?
- Should class Square be derived from class Rectangle; or vice-versa?

Thought Questions

- Suppose we have a class FacultyMember, which has methods

  hire()

  fire()

  Should we have a sub-class TenuredFacultyMember?

Thought Questions

- Suppose we have two kinds of Employee:

  Agent

  LotAttendant

  Should both of these be classes extending a common base class: Employee?

Dependency-Inversion Principle (Robert C. Martin)

- **Details should depend on abstractions; abstractions should not depend on details.**
- High-level modules should not depend on low-level modules; both should depend on abstractions.
- In other words, don't let low-level modules "call the shots" for high-level ones. The high-level ones are where policies should be set.
- Succinctly: ***Specify the interface first***, then implement.

Law of Demeter

Law of Demeter (LoD)

colloquial version

- "Only talk to your immediate friends".
- [not meaning *friends* in the sense of C++ classes and methods]
- so named by Prof. Karl Lieberherr, Northeastern University:
- LoD home page:
  http://www.cs.neu.edu/home/lieber/LoD.html

Law of Demeter Principle as stated by its author

- Each unit [i.e. class, method] should only use a limited set of other units: only units "closely" related to the current unit.

- Main Motivation: Control information overload. We can only keep a limited set of items in our short-term memory.

- Secondary Motivation: maintainability.

LoD: More Specific Interpretation

- An object should only invoke methods of:
  - objects that are ***declared*** within it
  - objects that are ***parameters*** *of the method*
  - ***itself***
  - objects that it ***creates***

- mnemonic DPIC (“depict”)

Precluded by LoD

- Do not extract object B from an object A and perform an operation on it.
- Instead, recast what you want to do as an operation on A. That operation may call operations on B.

Example of Violating the Law of Demeter

- class Patron
  {
  void sendNotice();
  }

```
library.getBook("Ulysses").getBorrower().sendOverdueNotice();
```

- class Book
  {
  Patron getBorrower();
  };

The above statement is considered bad:
The client of library has to *know about* books and borrowers just to send this notice.

- class Library
  {
  map<Book> booksByTitle;

Remedying the Violation

- class Library
  {
  map<Book> booksByTitle;

  void sendOverdueNotice(string Title);
  };

library.sendOverdueNotice(“Ulysses”);

Rumbaugh, Booch and the LoD

Rumbaugh: "Avoid traversing multiple links or methods. **A method should have limited knowledge of an object model.** A method must be able to traverse links to obtain its neighbors and must be able to call operations on them, but it should not traverse a second link from the neighbor to a third class."

Booch: "The basic effect of applying this Law is the creation of **loosely coupled classes**, whose implementation secrets are **encapsulated**. Such classes are fairly unencumbered, meaning that to understand the meaning of one class, you need not understand the details of many other classes."

Further Guidelines
related to LoD
will be seen when we
discuss “Design Patterns”