Object-Oriented Design II

Learning Goals

You are expected to be able to:

- ✓ **Perform a basic assessment of a design using the design principles of coupling and cohesion.**
- ✓ **Use a design pattern to solve a design problem with desired properties. In particular, learn how to use the design patterns of Composite, Observer, and Singleton.**

What is Coupling?

The activity of design involves considering many different alternatives to determine a design that makes appropriate trade-offs between such aspects as simplicity, flexibility, understandability and performance for the application under development. Software modules (e.g., classes in an object-oriented program) must depend upon each other to form a working system. While dependencies between modules are necessary, the more dependencies in a system, the more interconnected the modules and the harder it will be to change the system. *Coupling* is the term used in computer science to indicate dependencies between modules.

Consider the modules shown in Figure 1a. In this design, module A depends upon both module B and module C and module B depends on module C. If module C is changed, the software developer must then consider if module A and module B must be changed. In contrast, changes to module A are less likely to directly affect module B or C.

Figure 1b provides an alternate design involving the same modules. In this case, module A depends upon module B which in turn depends on module C. In this case, if module C changes, the software developer need only first consider if changes to module B are necessary. If no changes to module B are needed, the developer need not consider module A. The design in Figure 1b exhibits lower coupling (dependencies) between the modules than the design shown in Figure 1a.

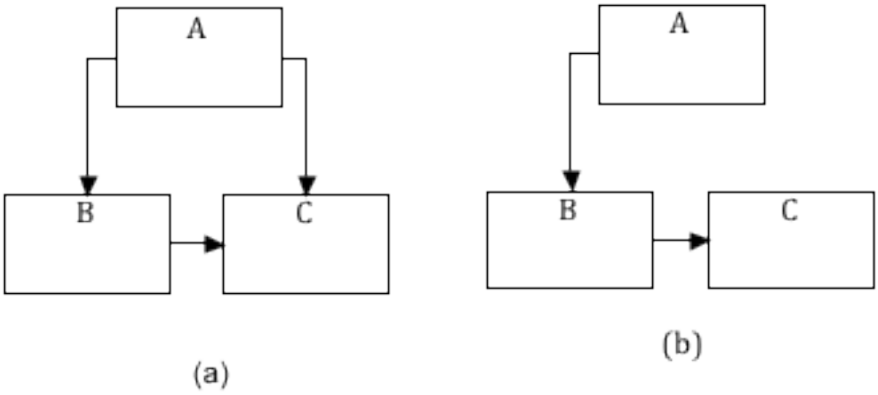

Figure 1: Alternate designs showing different degrees of coupling

If you read the Wikipedia page on software coupling (http://en.wikipedia.org/wiki/Coupling_(computer_science)), you will see that there are many different variations of coupling that have been defined. If you dig deeper on the web you will see that there are many different ways to calculate metrics about the coupling, which aim to provide a quantitative measure of the degree of coupling in a system so that designs can be compared based on these measures.

In this course, we are concerned with the meaning of coupling, why it is important and identifying where excessive coupling occurs in a design.

Read more in the following article:

Reducing Coupling by Martin Fowler, *IEEE Software*, July/August 2001, p. 102-104. This article is available either on-line through the UBC library system or through the following public link: http://martinfowler.com/ieeeSoftware/coupling.pdf.

IEEE Software is one of the main professional magazines that aim to keep software practitioners up-to-date with the latest in software technology.

What is Cohesion?

Cohesion is the term used to describe whether the functionality provided by a software module are related and focused. A highly cohesive class in an object-oriented software system contains methods that all pertain to providing the responsibilities of the class. A class with low cohesion would have methods that are not directly related to the responsibilities of the class.

Here is an example. Remember the `Animal` class from the Data Abstraction reading? This class represents an animal in a software system to support feeding of animals at an aquarium. Here it is again.

```
// An animal at the aquarium.
class Animal {

   // Construct an animal. We need to say what food the animal
   // eats
   Animal( List<Food> acceptableFoodToEat ){…}

   // What can this animal eat?
   List<Food> getAcceptableFoodToEat() {…}

   // When should the animal eat next?
   Date nextTimeToFeed() {…}

   // Remember what the animal last ate
   void recordLastFeeding(List<FeedingRecord> foodEaten) {…}
}
```

The above class exhibits high cohesion because all of the methods of the class pertain to the feeding of an animal.

In contrast, consider the version of the `Animal` class shown below also intended for use within the aquarium animal feeding system. This class exhibits lower cohesion as it is unclear why a method for printing a given photograph would be included in this class. Now this is a fairly obvious case of a method not fitting within a particular class. Often, it won't be so clear whether a method breaks the cohesiveness of the class or not. In assessing whether the methods form a cohesive class, you can consider the responsibilities as outlined in a responsibility-driven design for the class. Methods that decrease the cohesiveness of a class often creep in over time as developers place methods in a class that should possibly be put within a new class or as developers place methods to minimize coupling. Why would developers place methods in an inappropriate class? Sometimes it is because of a misunderstanding of the purpose of a class. Sometimes it is because it is at a point in the software development cycle where it is not possible to add a new class because of how the actual software product is built or deployed from the classes. Sometimes it is because the developer has the rights to change that class but not add a new one. Why would developers misplace methods to minimize coupling? Sometimes it is because the performance of the system will degrade by increasing coupling. Sometimes it is because one would have to include a lot more code in the software application by coupling to a class. These are just a few amongst many reasons.

```
// An animal at the aquarium.
class Animal {

   // Construct an animal. We need to say what food the animal
   // eats
   Animal( List<Food> acceptableFoodToEat ){…}

   // What can this animal eat?
   List<Food> getAcceptableFoodToEat() {…}

   // When should the animal eat next?
   Date nextTimeToFeed() {…}

   // Remember what the animal last ate
   void recordLastFeeding(List<FeedingRecord> foodEaten) {…}

   // Print a photograph of a visitor at the aquarium
   void printPhoto(Photograph photo) {…}

}
```

Classes that exhibit high cohesion are desirable as a developer can look at one class and reason about the abstraction the class provides. The less cohesive a class, the more a developer must find and visit multiple classes when making a change and the higher the coupling in the system as multiple classes end up interacting to provide desired functionality.

Assessing a Software Design

There are few hard and fast rules for assessing a software design. An obvious rule is whether a design supports the functionality needed by the application. As you saw in the Object-Oriented Design I reading, this assessment can be done by walking through scenarios of how the system is to be used and determining how the design supports those scenarios.

Designs are also assessed according to coupling and cohesion. Designs that exhibit low coupling and high cohesion are preferred. One particular aspect to look out for regarding coupling is whether there is more than one path of dependencies between classes. Sometimes multiple paths are necessary but in general, we want to have only one path between classes. Figure 1 provides an example of why one path is desirable. The *What is Coupling?* Section of this document provides an explanation. Similarly, cycles in the dependency graph are to be avoided.

Design Patterns

Coming up with great designs on your own is hard. Creating great designs requires experience to understand the ramifications of various design constructs on the performance

of the eventual system, the ease with which the system can be modified and other similar qualities. To help share good design practices, there has been effort made in capturing and documenting good *design patterns*. A design pattern is a reusable solution to a commonly occurring problem. (See the Iteration Abstraction reading for an introduction to design patterns). The most famous design patterns are the ones defined by the Gang of Four (Gamma, Helm, Johnson and Vlissides) [1]. We will look at some specific design patterns over the next few lectures. Here are pointers to readings about these design patterns. Please scan over the Wikipedia pages listed below before class and we will go through what these are in detail with examples in class.

1. Singleton pattern (http://en.wikipedia.org/wiki/Singleton_pattern)
2. Observer pattern (http://en.wikipedia.org/wiki/Observer_pattern)
3. Composite pattern (http://en.wikipedia.org/wiki/Composite_pattern)

References and Further Reading

Material in this reading is based on:

[1] Design Patterns: Elements of Reusable Object-Oriented Software by Gamma, Helm, Johnson and Vlissides. Addison-Wesley. (This book is likely available in the Dept. of Computer Science reading room.)