Cohesion and Coupling

Let’s consider the boundaries between classes—what functionality should be in one class vs. in another. Let’s start by considering this question:

What’s the harm of having too few classes in a program? Too many? [Submit your answer here.]

Maximizing cohesion

The basic guideline of class design is,

    Every class should be responsible for doing one thing only and doing it well.

Readers should be able to understand the behavior of the class without reading the code.

The fact that all the behavior of a class is closely related is called “cohesion.”

What’s a good example of a class with high cohesion?

What’s an unfortunate example of a class with low cohesion?
Another example is a “god” class that controls all the other objects in the program. The objects are reduced to mere data-holders.

**Separation of responsibility**

It is not always clear which class should do what. Sometimes we need to consider the advantages and disadvantages of each assignment of responsibility.

Consider these examples from Skrien.

*Example 1:* When an array of objects needs to be sorted, the objects need to be compared to each other.

- Should the objects know how to compare themselves to other objects with a method similar to *String’s compareTo(Object)* method, or
- should a separate object, such as the *Comparator*, be responsible for doing the comparing?

```java
public class Comparator {
    public int compare(String o1, String o2) {
        return s1.compareTo(s2);
    }

    public int compare(Integer o1, Integer o2) {
        int i1 = o1.intValue();
        int i2 = o2.intValue();
        return i1 - i2;
    }

    // ...compare methods for other types of data...
}
```
For example, the **Array** class in the `java.util` package includes two methods that sort arrays of objects. One method uses the `compareTo(Object)` method of each object and the other uses a `Comparator` to do the comparing.

**Example 2:** Consider a **LinkedList** implemented from **Nodes**.

Each **Node** consists of data and a link (**next**).

When your program traverses a list, which object is responsible for keeping track of where it is?

- The *client* could keep a reference to the current node, and dereference **next** to move to the next node.
- The **LinkedList** object could keep a reference to the current node
  Then the client would ask the list—
  - for the data in the current node, and
  - to move to the next node (causing the list to update its **current** pointer).
- A *third object* could keep track of where the program is in the list.
  Then the client would ask the third object—
  - for the data in the current node, and
  - to move to the next node (causing the 3rd object to update its **current** pointer).

Which approach is best? [Vote here.](#)

- What, if anything, is wrong with the first one?
- What, if anything, is wrong with the second one?
- What, if anything, is wrong with the third one?
Guideline 1: Different responsibilities should be divided among different objects.

Guideline 2: Encapsulation. One class should be responsible for knowing and maintaining a set of data, even if that data is used by many other classes.

Corollary: Data should be kept in only one place.

One class should be chosen to manipulate a particular type of data. Other classes must ask this class when they need to use or change the data.

Let’s see what happens if this guideline is not followed.

For example, suppose you have an object of class Department that is responsible for maintaining a collection of Employee objects, held in an ArrayList.

Other objects may need to access the Employee objects.

The Department would have a getEmployees method that returns the ArrayList of Employee objects.

What is wrong with this approach?

How can this risk be avoided?

1. Have the getEmployees method return an ArrayList of the Employees, but make it a new ArrayList that is a shallow clone of the Department's ArrayList.

2. Assuming that other objects rarely need all the Employee objects, have a “getter” method that finds and returns an employee specified by particular criteria.
3. Have the `Department` class manipulate the `Employee` objects. Clients have to ask the `Department` for any details regarding `Employees` that they need.

4. Replace the `getEmployees` method with an `iterator` method that returns an `Iterator` over the `ArrayList`.

The principle of encapsulation states that the `Department` should never let other classes see the actual `ArrayList`, but only the data in the `ArrayList`. See the text for several more examples.

---

**Guideline 3:** *Expert pattern.* The object that contains the necessary data to perform a task should be the object that manipulates the data. “Ask not what you can do to an object; ask what the object can do to itself.”

Guidelines 2 and 3 establish that data should not be manipulated in more than one place.

Similarly, code should not be duplicated in more than one place.

**Guideline 4:** *The DRY principle.* Code should not be duplicated. A given functionality should be implemented only in one place in the system.

Why is this a good guideline?

---

**Minimizing coupling**

Classes frequently need to be modified.

They should be written in such a way that changing one class is not likely to break other parts of the code.

**Guideline 5:** Design your classes so that they can handle change.
One way of doing this is not to hold data in variables of a specialized type.  
http://www.mosaicshades.com/antique2005/sockets/index.htm#par13b

The idea is to define your variables and values to have the widest possible type rather than the narrowest type.

The widest possible type in Java is an interface that can be implemented by any number of classes.

Guideline 6: Code to interfaces, not classes. Wherever possible write your code so that objects are referred to by the interfaces they implement instead of by the concrete class to which they belong.

Guideline 7: Encapsulate the concepts that vary.

What example have we seen of this today?

**The open-closed principle**

The open-closed principle can be expressed as follows:

*Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.*

Why is it harder to add new functionality to a program when an existing class needs to change?

Keeping the data of a class private helps assure that the class is closed for modification.

If classes are not to change, then you need to be careful to design them so they don’t need to. This suggests …

Guideline 8: Give classes a complete interface.

A class should have a full set of methods so that it is as reusable in as many situations as possible. Suppose we have a GUI component
that has a `setSelected()` method that highlights itself. What other method should it have?

The component should probably also have an `isSelected()` boolean function that tells whether the component is currently highlighted.

*Guideline 9:* A well-formed class has a consistent interface.

By “consistent,” we mean that the methods that do similar things should be laid out similarly.

Suppose a class maintains and manipulates an indexed collection of people. It has

- a method that sets the \(i\)th person’s name and
- a method that sets the \(i\)th person’s age.

The two methods should have similar names and similar arguments in the same order.

Suppose one signature is `setName(String name, int index)`. What should the other be?

**The Law of Demeter**

Long chains of method calls mean there is a large amount of coupling between classes.

Consider this approach to getting a bank balance:

```java
```

Assume that `b`, `r`, `c`, and `a` are all strings.

How many classes does the calling class need to know about?
Another way to handle this would be to code,

```java
Balance balance = atm.getBalance(b, r, c, a);
```

Now only the ATM class, not the caller, needs to worry about the existence of the branch, the customer account, etc.

The Law of Demeter says that a class should only send messages to

1. this object itself
2. this object’s instance variables
3. the method’s parameters
4. any object the method creates
5. any object returned by a call to one of this object’s methods
6. the objects in any collection that falls into these categories

It should *not* send messages to objects that are returned by calls to other objects.

This is also a good organizational principle. Consider what used to happen when I wanted to retrieve or send back homework to off-campus students. I just phoned Eva Boyce and she took care of it …

**Give an example** of a scenario that (a) follows the Law of Demeter, or violates it.

**Factory Method design pattern**

Factory Method is the first *creational* pattern we’ve seen—that is, the first pattern that is used for creating objects.

As we know, in o-o languages, objects can be created with some kind of `new` method.

Calling `new` is fine if the calling code knows what kind of object it wants to create. But a lot of times it doesn’t.

Suppose, for example, that the code is copying a diagram, which consists of *Shapes*. If it gets in a loop and copies the shapes one by
one, you’d need a big \texttt{if} statement to decide which kind of object to create.

Well, better to \texttt{encapsulate that logic in a creation method}, rather than to expose it at the call site.

Then the client only needs to get in a loop calling the \texttt{getShape()} method, and each time, the right kind of object will be created.

The method that creates the object could even be a \texttt{static method} of the class that returns an instance of that class. This has two advantages over using constructors:

1. The “new” object might in fact be a reused object that was previously created (think “buffer pool”).

2. The object that is created might actually be a subclass object.

In any case, the client “is totally decoupled” from the code that creates the object.

Put your answers to the exercise \texttt{here}. 