Cohesion and Coupling

Outline for Week 12

I. Cohesion & coupling
   A. Maximizing cohesion
   B. Separation of responsibility
   C. Minimizing coupling
   D. The Law of Demeter
II. Creational patterns
   A. Factory Method
   B. Abstract Factory

Let's consider the boundaries between classes—what functionality should be in one class vs. in another.

We want to maximize *cohesion* and minimize *coupling*.

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."

Another example is a "god" class that controls all the other objects in the program. The objects are reduced to mere data-holders.

A common problem is when code to check for a condition is littered throughout the system, so that to understand a class, the reader needs to read about several unusual conditions. Here is an example from Expertiza.

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.

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?

```
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...
}
```

Submit your answer here.

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`**.

Guideline 3: *Information Expert pattern*. Assign a responsibility to the class that has the data needed to fulfill that responsibility.

“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.

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.

Here is an example of an e-mail sender that comes in a high-coupling and low-coupling package.

Look at the difference between the two packages and answer these questions.

Guideline 6: Do not use class methods when instance methods will suffice.

What are some ways that class variables or methods can be used?

Overuse of class methods increases coupling between classes.

Functionality that should be a method of one class is in another class, where no object is a receiver. This means that some of the responsibilities of the class are actually implemented in other classes.

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:

```
Balance balance = atm.getBank(b).getBranch(r).
getCustomer(c).getAccount(a).getBalance();
```

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,

```
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 happens when I wanted to retrieve or send back homework to off-campus students. I just phoned Eva Boyce and she took care of it …

Here is an exercise on the Law of Demeter.

Creational patterns

Factory Method design pattern

Factory Method is a *creational* pattern—that is, a 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 **`Shape`**s. If it gets in a loop and copies the shapes one by one, you'd need a big **`if`** statement to decide which kind of object to create.

Well, better to 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 **`getShape()`** method, and each time, the right kind of object will be created.

The method that creates the object could even be a **`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.

Choose one of the following two exercises for postcodes.

- An example on creating wifi or Bluetooth streaming connections.
- An example that creates postcodes for the US, India, and the UK.

The Abstract Factory design pattern

The Factory (or Factory Method) pattern is good for creating single objects of a specific type, as long as those objects don't have to "match" any other objects.

But a lot of times we do need objects to "match."

- User interfaces need all of their widgets to have the same "look and feel."
- Web (or print) pages have a certain style, and all the elements on the page need to match that style.
- A language run-time environment needs to make the appropriate calls to the operating system it is running on.

In all these cases, instantiated objects all need to be from the right "family."

My favorite example of this comes from Refactoring Guru. When a shop sells a set of furniture to a buyer, it's important that the set "match."

Let's look again at the entities in the pattern.

- The *client*—the code "using" the pattern—has-an Abstract Factory.
- The Abstract Factory is an interface that contains a method for creating each different "product."
- One or more Concrete Factories implement the Abstract Factory interface.
    - Hence, each concrete factory implements a method for creating each different "product."

- When the various `createProduct` methods of a particular Concrete Factory are invoked, they create objects that “match.”
    - Specifically, the objects “match” because they implement the variant defined by the same concrete factory.

Now, to make sure you are following, answer these questions about the furniture example.

- What is a “modern sofa”? An abstract product, concrete product, etc.?
- What is a “chair”?
- What is a “Victorian furniture factory”?
- What does the method `createCoffeeTable` return?

Again, there are two choices of exercise:

- An example that builds applications in the Semantic or Angular framework
- An example that creates phone numbers and postcodes for the US or the UK.