CPS122 Lecture: Cohesion and Coupling

Last revised May 29, 2009

Objectives:

1. To introduce cohesion and coupling as criteria for evaluating designs

Materials:

1. Cohesion/coupling exercises worksheet

I. Introduction

A. As you are doing design, it is important to have criteria in mind for evaluating the quality of the design.

B. Today, we look at two such criteria: cohesion and coupling.

1. In a good design, the various component parts (e.g. the classes) have high cohesion.

2. In a good design, the various component parts (e.g. the classes) have low coupling

II. Cohesion

A. We say that an entity is *cohesive* if it performs a single, well-defined task, and everything about it is essential to the performance of that task.

(Note that we have defined cohesion in terms of an entity. The cohesion test can be applied to classes (as is our primary focus here); but it can also be applied on a lower level to the individual methods of a class (each of which should be cohesive) or, on a higher level, to a package of related classes or an overall system or subsystem.)

1. An important goal in design is to try to ensure that each entity we design (class, method, system) exhibits the highest possible level of cohesion.

2. A good test for cohesion is "can I describe the purpose of this entity (class, method, etc.) in a short statement without using words like 'and'?" (This is one of the benefits of writing a prologue comment for a class or method before you write the code - it helps you to think about whether what you are about to produce is truly cohesive.)

B. Over the years, software engineers have identified various sorts of possible cohesion, which can be ranked from most cohesive (good) to least cohesive (bad).

Unfortunately, different writers list different of types of cohesion, and use different names. Of course, the ultimate task is not to determine what kind of cohesion a given entity exhibits, but rather to produce the most cohesive entity possible.

Here is one approach:

1. Desirable sorts of cohesion

   a) Functional cohesion - the entity performs a single, well-defined task, without side effects. [A well-designed method will exhibit this.]

   b) Informational cohesion - the entity represents a cohesive body of data and a set of independent operations on that body of data. [A well-designed class will exhibit this.]

2. Less desirable sorts of cohesion - listed in decreasing order of desirability

   a) Communicational, Sequential, Procedureal cohesion - the entity is responsible for a series of tasks which must be performed in some order. (With fine distinctions between communicational, sequential, and procedural that we will omit here)

   b) Temporal cohesion - the entity is responsible for a set of tasks which must be performed at the same general time (e.g. initialization or cleanup)

   c) Logical - the entity is responsible for a set of related tasks, one of which is selected by the caller in each case.

      In a case like this, it may be better to have several kinds of entity - perhaps using polymorphism.

   d) Utility cohesion - the entity is responsible for a set of related tasks which have no stronger form of cohesion.

      Example: the java.util package and the java.Math class - the fact that a level of cohesion is less desirable does not mean it can always be avoided!

3. Undesirable: coincidental cohesion - the entity is responsible for a set of tasks which have no good reason (other than, perhaps, convenience) for being together.

C. For any entity that has less than the highest possible cohesion, it is worthwhile considering whether its cohesion can be improved.

1. Sometimes, this is a simple rethinking of its purpose statement. If the purpose contains "ands", it may be possible to construct a statement that implicitly includes all the items and nothing else.

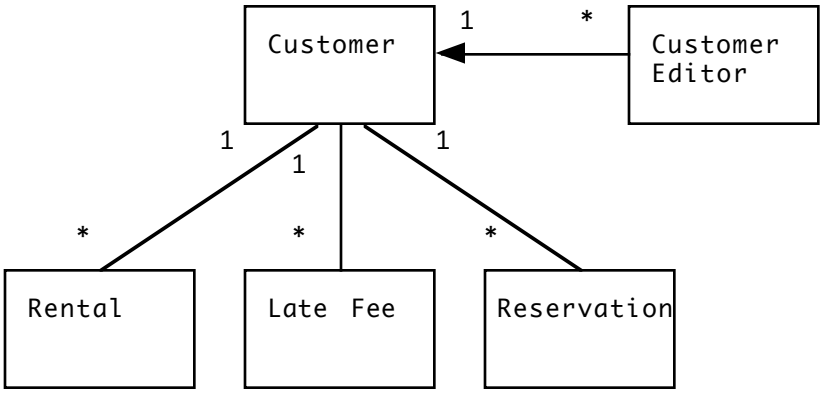
   Example: Consider a method that enrolls a student in the course. Its purpose statement might read: "Add student to course and add course to student". A simple rephrasing might be "Enroll student in course" that implicitly includes both of these, and improves cohesion..

2. Often, though, when an entity has low cohesion, it may be possible to refactor the design to produce higher cohesion by splitting the low cohesion entity into two or more entities with higher cohesion.

   a) For example, some entity may be responsible for two different kinds of tasks, but can be refactored into two associated entities, each of which is responsible for one of the kinds of tasks.

   b) One place where this often occurs arises when you have a class that represents some entity that can be displayed and perhaps edited in a GUI. In this case, it may make sense to refactor the design into a class that has the responsibilities associated with representation and another related class that has the GUI responsibilities.

   Example: One might create a Customer class in the video store project that is responsible for representing a customer (including keeping track of the customers rentals, late fees, and reservations) and also for GUI display/editing of the customer. It is probably better to create two classes:



(Something like this may be worthwhile in iteration 2, for adding a new customer and likewise for title.)

c) Another example of this occurs in the videostore "gift" GUI which has a `RentDetailsCard` plus a `RentDetailsCard` (and may have a `Rental` class, or may just represent in `Customer` and `RentableItem`).

D. Next class, we will discuss some handout exercises

Give out worksheet

III. Coupling

A. Coupling is a measure of the extent to which an entity depends on other entities. We will discuss coupling in terms of classes today, but (as with cohesion) coupling can also be considered at other levels.

B. A system has low coupling just when the various component parts have minimal dependency on each other. Of course, some coupling is essential - else you have a society of hermits. But what we want is to eliminate unncessary coupling. This makes modification/maintenance of the system much easier.

C. Recall that a class A depends on another class B if A:

1. A has an association with that B (with navigability toward B if unidirectional)..

2. Generalizes or realizes B

3. Has a dependency on B - through methods that

a) Have local variables of type B

b) Have parameters of type B

c) Have a return value of type B.

4. Dependencies have two important consequences:

a) If a class A depends on a class B, and we want to build a system that reuses class A, then we must also include class B in the system, whether or not it would otherwise be needed.

b) If a class A depends on a class B, and class B is modified, class A may need to change as well.

5. While dependencies are unavoidable (and indeed often necessary), what we want to do is to minimize the likelihood of casading modification occurring, which depends on the strength of the coupling between two classes.

D. While it is classes that are coupled to one another, it is typically in the methods of the dependent class that one can take measures to reduce (or even sometimes eliminate) the coupling, as we shall see below.

E. As was the case with cohesion, software engineers have developed some categories of coupling. Here is one approach (ranked from most lowest - therefore most desirable - to highest)

1. Data coupling occurs when a method of class A has parameters (or local variables or a return value) of class B and uses the class B object as a single, atomic piece of data. This ususally can't be improved.

2. Stamp coupling occurs when a method of class A has parameters (or local variables or a return value) of class B and depends on the structure of the B object (i.e. uses part of it).

a) Example:

Suppose we have a class `Person` with a method called `getBirthDate()`. Suppose we now want to create another class `DriversLicense` with a method called `isJuniorOperator()` (which returns true if an individual is under 18). One way to structure this would be as follows:

```
boolean isJuniorOperator(Person p)
{
   Date birthDate = p.getBirthDate();
   // return true if birthDate is less than 18 years
   // before today's date
}
```

However, if we changed the `getBirthDate()` method of `Person` in some way (e.g. renamed it or changed the way we stored information about the driver), we might also have to change the `isJuniorOperator()` method of `DriversLicense`.

b) Often, stamp coupling can be reduced by rethinking the parameters of a method. For example, in this case we could design the method to just take the person's birthdate (all it needs), rather than the whole person as a parameter.

```
boolean isJuniorOperator(Date birthDate)
{
   // return true if birthDate is less than 18 years
   // before today's date
}
```

Now we rely on the caller to extract the necessary birth date information from the `Person` object, reducing the coupling between `DriversLicense` and `Person`, since we no longer need know that a `Person` explicitly stores a birth date or provides a `getBirthDate()` method to get it. The effect of this is actually to exliminate the coupling between `DriversLicense` and `Person` in this case.

3. Control coupling arises when a method does different things depending on the value of a "flag" parameter.

   a) Example: in your video store, you might eventually create a method like this:

   ```
   updateCustomer(int whatKind, Customer customer)
   ```

   where `whatKind` takes on the values `ADD`, `EDIT` or `DELETE`, and `customer` is used for `EDIT`, but is not used at all for `ADD`, and only the id is used for `DELETE`.

   b) Often, control coupling can be reduced by replacing the method with multiple methods - e.g. (in the example):

   ```
   addCustomer() ...
   editCustomer(Customer customer) ...
   deleteCustomer(String customerId) ...
   ```

   (In this particular example, this also has the effect of producing cohesion, and eliminates stamp coupling in one case)

4. Common coupling arises when a method depends on a global variable. This is less common in OO systems, but can still occur.

   a) Example: In the video store problem, the rental rate can be set by management. One might address this by including a variable

`rentalRate` in the the class `StoreDatabase` - in which case code like the following might occur:

```
amountDue += StoreDatabase.rentalRate;
```

This is undesirable, because any change to the variable (e.g. giving it a different name or changing its type can "break" other classes - sometimes in startling ways (like charging someone $3000 for a rental, because the representation for `rentalRate` was changed from a float representing a dollar amount to an int representing a number of cents!)

b) Than be fixed by using encapsulation, with a method like `getRentalRate()`. Where should this method occur?

ASK

(1) It is tempting to put it in `StoreDatabase`.

(2) However, since movies and games have different rental rates, it would be necessary to include some sort of specification as to which kind of item is being rented (a form of control coupling), Lower coupling results from putting such a method in `Movie` and `Game`, (and as an abstract method in `Title`), with each calling either `getMovieRentalRate()` or `getGameRentalRate()` in `StoreDatabase` as appropriate.

5. Content coupling occurs when a module surreptitiously depends on the inner workings of another module.

(It turns out its almost impossible to illustrate something this bad using java!)