CPS122 Lecture: Cohesion and Coupling

Last revised March 27, 2023

Objectives:

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

Materials:

1. Projectables
2. Types of Cohesion/Coupling on Canvas
3. Cohesion/coupling 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 <u>high</u> cohesion.

      2. In a good design, the various component parts (e.g. the classes) have <u>low</u> 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 <u>purpose</u> 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:

Show Types of Cohesion/Coupling on Canvas - focus on cohesion side

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, Procedural 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 the method in the UML labs that enrolls a student in a course (`doEnroll()`). 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 `Patron` class in the library project that is responsible for representing a patron (including keeping track of the patron's checkouts, fines, and reservations) and also for GUI display/ editing of the patron.

      This might show in a CRC card for the class like this:
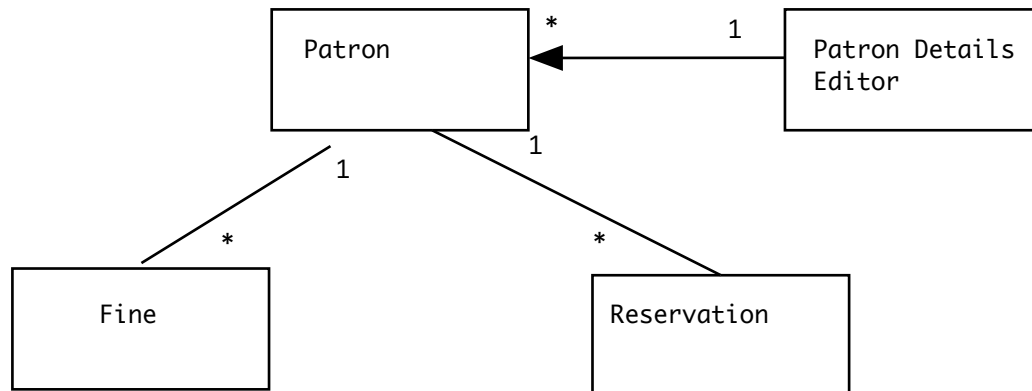
      Class Patron

Responsibilities	Collaborators
Keep track of basic information	
Provide GUI facilities for editing	
Keep track of checkouts	Copy
PROJECT	

It is better to create two classes, splitting the responsibilities

```
                              *                    1
   ┌──────────────────┐              ┌──────────────────────┐
   │     Patron       │◄─────────────│   Patron Details     │
   │                  │              │   Editor             │
   └──────────────────┘              └──────────────────────┘
         1          1
        /            \
       *              *
┌──────────────┐    ┌──────────────┐
│    Fine      │    │ Reservation  │
│              │    │              │
└──────────────┘    └──────────────┘
```

(An approach like this is presumed for iteration 2 for adding a new patron and likewise for adding the two types of Item.)

PROJECT

D. Give out worksheet; allow class to do Cohesion portion in groups, then discuss.

My answers: (PROJECT)

1. D - the only thing these operations have in common is the fact that they're done at the same time.

2. A - this is a good example of a module that performs a single, clearly-defined task

3. C - the cohesion arises from the need to perform steps in a certain order.

4. B - this is a good example of a cohesive class

5. G - the lack of cohesion here is about as bad as it can get!

6. E - Probably it would be better to have separate drawCircle() etc. methods - which would also make it easier to extend the functionality - or to create a class Shape with method draw() and subclasses for different shapes that define draw() appropriately,

III. **Coupling**

A. Coupling is a measure of the <u>extent</u> 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 <u>unnecessary</u> 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 usage 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 cascading modification occurring, which depends on the strength of the coupling between two classes.

D. While it is <u>classes</u> that are coupled to one another, it is typically in the <u>methods</u> 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)

Refer to other side of handout

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 usually 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:

```
/** Test to see whether a driver is classed as
 *      a junior operator (< 18 years old)
 *      @param p the driver to be checker
 *      @return true if a junior operator
 */
boolean isJuniorOperator(Person p) {
    Date birthDate = p.getBirthDate();
    // return true if birthDate is less than 18 years
    // before today's date
}
```

PROJECT

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
}
```

PROJECT

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 <u>eliminate</u> 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 library project, you might eventually create a method like this. (Not encouraged!)

```
updatePatron(int whatKind, Patron patron)
```

PROJECT

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

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

```
addPatron() ...
editPatron(Patron patron) ...
deletePatron(String phoneNumber) ...
```

PROJECT

(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 or constant. This is less common in OO systems, but can still occur.

Example: In the library problem, one would need to know the fine rate per day overdue. One might address this by including a constant `FINE_RATE` in the class `Fine`. Now suppose that the value of this constant is changed. Obviously, the class `Fine` would need to be recompiled. But the class `ReurnUseCase` would also need to be recompiled if it uses this constant in calculating the fine amount.

This could be avoided if the class `Fine` contains a method that does the actual calculation based on the value of this constant, which `ReurnUseCase` simply calls. Now `Fine` is the only class that needs to be recompiled if the constant is changed. (Even better might be

to make this a variable that can be set by management so nothing needs to be recompiled to change it!)

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!)

F. Allow class time to do coupling portion of worksheet in groups, then discuss.

My answers: (PROJECT)

1. D (The java equivalent of the global variable found in some programming languages is a static variable of some class)

2. B (because we pass in the whole person object, not just the one piece of information we need - hence the calculateAge method needs to know about the interface of Person - e.g. that it has a method to access the individual's birth date (getBirthDate() or the like. This coupling would not occur if the parameter to the method were the birth date - then it could be applied to any birth date, not just the birth date contained in a Person object.)

3. E (It's not actually possible to construct something this bad in Java!)

4. A - there is only one parameter, and it's a simple data item

5. C - one of the parameters is a flag that controls whether the method prints today's date.