Lecture Notes,

CS345, Winter 2018

Coupling and Cohesion

Introduction

High-level software design, meaning anything (much) above the design of a single method, consists primarily in separating the software into components, assigning responsibilities to those components, and specifying the interfaces between components. And, doing this repeatedly until the components get "small enough." A big question we must answer is what components, functions, responsibilities to put together and what to separate.

The answer to this question can be found by observing that we want to make it as easy as possible to understand, test, modify (fix and enhance), and reuse our software.

Based on this we want to:

- Keep similar code close together – make the software easy to understand and modify,
- Keep things that have to change together as close together as possible – make the software easy to modify, minimize the scope of bug fixes,
- Minimize duplication in the code – make the software easy to modify – a change only has to be made in one place, and
- Keep components that will be reused together close together – make the software easy to reuse.

Software engineers use the term **cohesion** to refer to the extent to which components are closely related to each other.

We also want to:

- Keep unrelated code in separate components – avoid having to think about multiple concepts simultaneously,
- Keep changes in one component from requiring changes in an "unrelated" component,
- Keep components that will not be reused together separated and disconnected.

Software engineers use the term **coupling** to refer to relationships between software components. That is, two components are **coupled** when understanding, changing, or reusing one component may require you to understand, change, and/or reuse another.

In general,

- Cohesion is good – you should work to increase cohesion and separate non-cohesive components into multiple components.
- Coupling is bad – you should work to decrease coupling.

Note that even though coupling is not desirable, some level of coupling is unavoidable. So, we work to minimize coupling and to convert from more to less obnoxious forms of coupling.

Kinds of Cohesion

In general, in O-O design, each class should have a small cohesive set of responsibilities. That will guarantee that the methods in the class have good cohesion. However, we still have to consider how to organize individual classes into packages, subsystems, etc.

Software engineering researchers have identified a number of different kinds of cohesion in software components. Here are a few ordered from strongest to weakest (best to worst):

- Functional – components are grouped together because all components contribute to a single well-defined responsibility or task.
- Sequential – components are grouped together because the output of one is input to another.
- Communicational – components that operate on the same data are grouped together. Note that sequential and communicational cohesion are based on grouping together components that share an interface.
- Procedural – components that perform the same sequence of operations are grouped together.
- Temporal – components are grouped by when they are processed. If you have a system that inputs data, analyzes data, and outputs data, then input software, analysis software, and output software would be a temporal organization.
- Logical – components are grouped because they are logically categorized as doing "the same thing". The Java Math package is a good example of this.
- Coincidental – components are collected together due to some arbitrary distinction such as common developer skill set or to avoid small modules. Many large systems will have a miscellaneous subsystem that was created simply to avoid small modules.

When designing software, you can think about the relationships between pairs of components and ask what kind of relationship(s) tie the components together. Are these relationships strong relationships (Functional, Sequential, Communicational)? Are they weak relationships (Logical, Coincidental)? Is there some other organization that would convert weak relationships into strong ones?

Also remember that as a software developer you have to understand both the tools and technology that are being used to build the software and the application that you are supporting. Grouping components together in order to minimize the required developer knowledge helps make your development more efficient.

Kinds of Coupling

Software engineering researchers have also identified a number of different kinds of coupling. Here are a few ordered, approximately, from strongest or tightest (worst) to weakest or loosest (best):

- Content coupling, also known as pathological coupling – one component modifies or relies on internal data of another. Note that this can occur even in languages like C++ and Java that enforce notions of privacy.
- Common coupling, also known as global coupling – two components share a global variable (as opposed to a global constant). In Java a global variable is a public static non-final attribute. Note that in most modern languages, common coupling is avoidable.
- External coupling – a component uses an externally supplied component which could be a library, some third party package, another system, etc.

- Type use coupling – multiple components use a globally defined data type. One form of this is subclass coupling where one class is a subclass of another. (A subclass is also coupled to its superclass.) In Java, we use interfaces as a mechanism to avoid being coupled to specific classes. Note that this transfers the coupling from the class to the interface.
- Control coupling – one component controls the flow of another, for example, by passing flags that direct what-to-do.
- Stamp coupling – components share a common data structure but only use a part of it, possibly different parts.
- Import coupling – one component imports another.
- Data coupling – components share data through parameters. This is distinguished from message coupling, below, by having one component save the value of a parameter so that it can potentially modify it later.
- Message coupling – components interact solely through parameters or message passing.

Again, when designing software, do you have tighter or looser forms of coupling? Can you convert, for example, common coupling into stamp coupling or type use coupling to data coupling.

Approaches to reducing coupling

Design Patterns

Many design patterns work to reduce coupling. Here are some examples:

- Bridge – reduce coupling between a behavior from its implementation
- Observer – eliminate coupling from an observed class to its observer
- Strategy – decouple a behavior from the specific algorithm(s) implementing that behavior
- Iterator – prevent using software from being coupled to the implementation details of a collection
- Builder – decouple construction of a complex object from its representation
- Command – decouple the details of a command from decisions about when and under what conditions the command is executed

Avoid Inappropriate Intimacy

Inappropriate intimacy is when a method in a class has knowledge of the implementation of another class.

Example: If I have a business application, classes in that application must have knowledge of the business logic (e.g. you can't withdraw money from an account if it would be overdrawn) and retrieving and updating data that is maintained by the application in a database. These two aspects of the application should be separated so that, for example, changes in how the database is accessed don't require that the classes containing the business logic have to change.

Tell, Don't Ask

The proper way to withdraw money from a bank account is to tell the account object to withdraw it and have the account object tell you whether it will or won't work. A wrong way is to ask the account object if it can be withdrawn and then tell it to withdraw the money. The worst way is to have the user interface object and the account object both implement the business logic required to determine whether or not you can withdraw money from the account.

Say it Only Once

Avoid code duplication. This is a guideline. However, if you start saying the same thing three or more times, you should start looking to see how to eliminate the duplication.

Law of Demeter

Only talk to you immediate friends.

If "you" are a method in a class, your immediate friends are:

- your class
- your parameters
- any object created by you
- any component objects of your class
- any global variable accessible by you.

Problem: Following this strictly leads to classes with very wide (large) interfaces since classes have to provide interfaces to methods on their components.

In general, this is only a guideline. But, be aware that when you expand the "friends" of a method that you are introducing additional coupling into your software.

Example of Law of Demeter

Given:

```
class A {
    public int getX() {/* return some int */}
}

class B {
    private A a;
    public A getA() { return a; }
}

B b;
print(b.getA().getX());
```

The Law of Demeter says that the print statement violates the Law of Demeter, because A is not an immediate friend of the code containing the print statement. Note: This example also shows why the Law of Demeter is sometimes known as the "two dot rule".

The Law of Demeter would rewrite the example as follows:

```
class A {
    public int getX() {/* return some int */}
}

class B {
    private A a;
    public int getAX() { return a.getX(); }
}

B b;
print(b.getAX());
```

In general, people do not observe the Law of Demeter. However, in the example above, the fact that B has a getA method should imply that the fact that B HAS-A A is an intrinsic part of the behavior of B. If it is not, that is, the fact that B HAS-A A is part of the implementation of A, then the Law of Demeter provides a way to eliminate A from the interface of B.