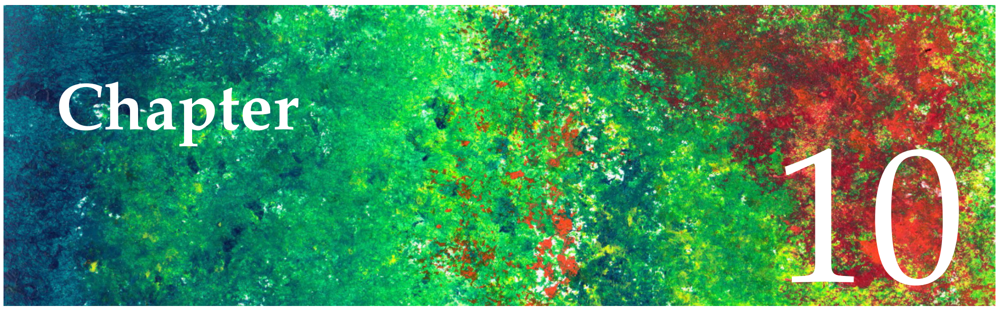

Coupling and Cohesion

Learning Objectives

Underlying almost all design patterns is a wish for maintainability and flexibility. The objective of this chapter is to introduce two qualities that are relatively easy to judge or measure in our code: *coupling* and *cohesion*. These two qualities are interesting because if I achieve high cohesion and low coupling then maintainability and reliability increase. I will also present a concrete rule, *law of Demeter*, that aims at getting low coupling.

10.1 Maintainable Code

Chapter 3, *Flexibility and Maintainability*, introduced the ISO definition of maintainability, its subqualities, and argued that maintainability measures cost of software change. Furthermore, *flexibility* was defined as a special case of changeability, namely the capability to add/enhance functionality purely by adding software units. The coding practice *change by addition, not by modification* is of course a means to achieve flexible software.

The question is what does maintainable code look like? The problem with the definition of maintainability is that it is not operational: it does not tell me how to write maintainable software. It merely tells me a label, *not very maintainable*, to put on my source code when I later discover it is highly expensive to modify to accommodate my customer's new idea. Of course, I want some more operational concepts, concepts that I can readily apply *while* I am programming in order to keep the code maintainable. And fortunately there are. In the next sections, I will cover some central qualities that are closely linked to source code and therefore relatively easy to measure or judge, and that have a huge impact on how maintainable the system will become.

10.2 Coupling

Definition: **Coupling**

Coupling is a measure of how strongly dependent one software unit is on other software units.

As usual, a *software unit* may be of any granularity: classes, subsystems, packages, even individual methods in a class.

Dependencies between software units come in many forms. In an object-oriented language several types of dependencies come to our mind: methods in a class are coupled by their dependency on the class' instance variables, and classes are coupled as they have relations with each other. Such relations may at the code level be in the form of object creation, method calls to another class, receiving an object as parameter in a method call, etc. Dependencies also exist between packages, when a class in one package calls a method in a class in another, etc. Common to these is that they are directly visible in our source code and you can make tools to find them and thus measure the coupling.

However, dependencies can be much more subtle and less directly visible in the code. This is the reason, you have to review the unit and subjectively judge the degree of coupling, as tools can only find some of the couplings between software units.

Exercise 10.1: List dependencies between software units that exist but are not described by method calls, object references, or other constructs that are easily spotted in the code.

Consider two packages whose dependencies are programmed in two different ways as outline in Figure 10.1. In the left side (a) the *system.gui* package contains two classes that depends on two, respectively three, classes in the *system.domain* package. Thus a change in the dark gray class in the lower package will most likely induce changes to be made in three other classes, those marked light gray in the figure. For instance, a method signature may have to be changed in the dark gray class which means that all places in the light gray classes where the method is called has to be reviewed and analyzed to make the classes compile again. Even worse, if a developer on the domain code team responsible for the *system.domain* package makes changes to what a method does in the dark gray class but does not change any method signature, he may remember to tell his colleague working on the light gray class within the same package, but perhaps he does not tell the GUI team down the hallway. Having strong test cases may catch such changes in the contract, but still effort has to be invested in making the light gray classes work correctly again. The packages have *tight* or *high* coupling; and high coupling lowers maintainability.

High coupling also impacts reliability in a negative way, as any defect or even just changed behavior introduced in the dark gray class may create undesirable effects and maybe defects in all classes depending on it. Thus even a small change in the class may lower reliability in many dependent classes. Also a class that depends on a lot of other classes becomes harder to read and understand again increasing the likelihood of introducing defects. Finally, a class that has many dependencies is harder to reuse in another context.

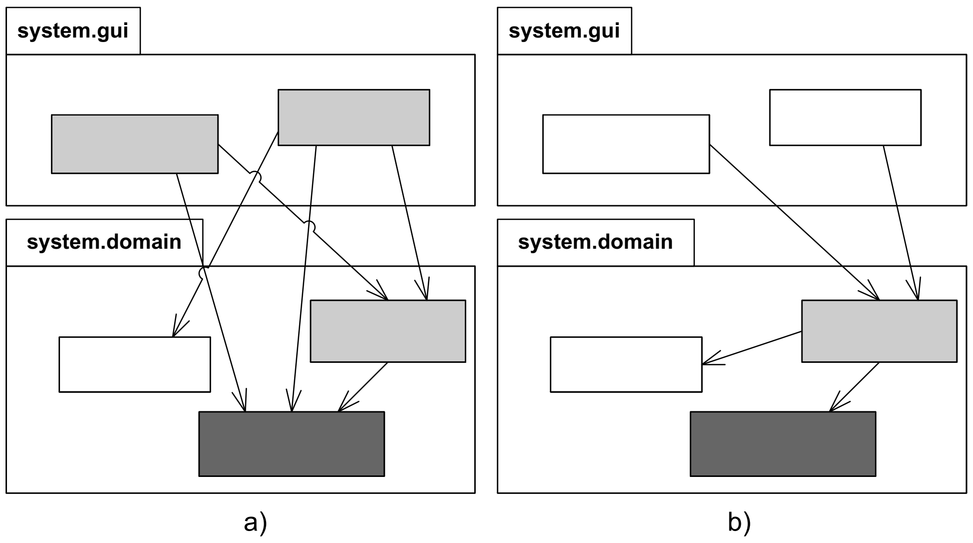

Figure 10.1: Tight (a) and low (b) coupling.

The implementation shown on the right side (b) of the figure shows the same two packages and the same number of classes, but attention has been paid to lower the coupling between classes. Thus only one class directly depends on the change marked class. The two classes in the upper package only depend on a single class in the lower package, in essence this class encapsulates the package (which is in fact the FACADE pattern, discussed in Chapter 19.) The packages have *weak* or *low* coupling.

Weak coupling generally works in favor of maintainable and reliable software. Maintainable because the change in the change marked class only has implications in a single dependent class—thus the effort to review, recode, debug and test is lower compared to the tight coupled version. And the side-effects and defects introduced in the dark gray class ideally only have reliability consequences for the single dependent class. It is only the interplay between these two classes that we must ensure the reliability of. Thus weak coupling also works towards more reliable software. And, a class with few dependencies is easier to reuse and make to work in a new software context.

10.3 Cohesion

Definition: **Cohesion**

Cohesion is a measure of how strongly related and focused the responsibilities and provided behaviors of a software unit are.

Cohesion is a term that basically means "being organized". Consider that all clean clothes of a family with a couple of children are stored in one big pile. It would take ages to find a pair of trousers and a pair of matching socks. Instead, if each member of the family has his or her chest of drawers, and each drawer contains one or two types of clothes, finding what to put on today is easy. High cohesion means placing the same type of clothes together in the same drawer. In software, it means software units should have just a few closely related responsibilities. As an example consider this remove method in a collection that one of my students wrote:

```
public void remove(Item i) {
  if ( list.isEmpty() ) {
    System.out.println(``The list is empty, cannot remove'');
  }
  [ remove behavior implemented here ]
}
```

When I read a class that must act as a collection I expect to see methods like add, and remove, and these names convey a lot of understanding of the behavior they provide. In the remove method above, however, the behavior is that of removing an item *and* occasionally printing something! These two behaviors are absolutely not related: removing items and printing. The name of the method does not adequately convey what it actually does. Cohesion is low.

Cohesion is a quality to strive for at all granularities. As an example of cohesion at the package level, consider the following two Java packages, organized by two different principles:

- Package *ABCClasses*: Contains all classes whose names begin with either the letters A, B, or C in my flight reservation system.
- Package *SeatBooking*: All classes related to booking a seat on a plane in my flight reservation system.

Clearly, package *ABCClasses* has low cohesion: it may contain all types of functionality and there is no clear relation between the responsibilities it serves. In contrast, package *SeatBooking* has high cohesion as seat booking embodies a small set of clearly defined responsibilities and the package provides behavior for just that and nothing else.

Exercise 10.2: Find an example of a high cohesion and a low cohesion package in the Java libraries.

High cohesion also contributes towards reliability and maintainability. Cohesive software units greatly increases the analyzability of software, the capability to find defects or parts to be modified, just as the organized approach for storing clothes allows us to find socks quickly. If a software unit has too many responsibilities it will get changed too often. If a class only implements a single feature it will change much less frequently, thus increasing stability.

Of course, you must keep your balance. Cohesive software does not mean software where every class only has one method, and every package only one class. In the pay station case, I started out with a cohesive abstraction, represented by the PayStation interface, but it had five–six responsibilities. However, they all were highly related to the “concept of being a pay station”. When need arose, it was refactored to introduce the rate strategy which is of course also very focused.

Cohesion and responsibility are two sides of the same coin per definition. Therefore the discussion on roles, responsibilities, and behavior in Chapter 15 is intimately related to the concept cohesion.

10.4 Law of Demeter

A very concrete rule that addresses coupling is the law of Demeter first formulated by Lieberherr and Holland (1989). This rule states

Definition: Law of Demeter

Do not collaborate with indirect objects.

To illustrate what "indirect" and "direct" objects means, let me start with an example. Consider that the Dean of the Faculty of Science is worried about low intake of students at the Department of Computer Science. The department is organized with a head of department that is assisted by several committees which in turn are also headed by a person. One of these committees is the Teaching Committee which is responsible for the webpages that provide information for future students interested in computer science. The actual editing of the webpages is performed by a secretary. As the webpages are confusing and badly written at the moment, one plausible action to take is to improve them. So the question is what action should the dean take? One highly infeasible action is to ask the department head to tell him who is the head of the teaching committee, and next ask him or her about who is the secretary responsible for updating the public relations web pages, and finally tell the secretary detailed instructions about what to change. In pseudo code this action would look something like this:

```
getHeadOfDepartment(CS).getHeadOfCommitte(Teaching).getSecretary().
  changeSection(webpage, section, newText);
```

This is absurd in any organization. The CEO or manager does not have the time to tell each individual employee what to do in detail several times a day. The feasible action of the dean is to tell the department head to make the necessary actions to increase student intake, and leave for him to decide what to do down the chain of command.

```
getHeadOfDepartment(CS).increaseStudentIntake();
```

In the terminology of law of Demeter, the department head is a *direct object* while the secretary is an *indirect object*, and the law states that the dean should talk to his department head, not to the secretary nor to the head of the teaching committee. The rule is also known as *Don't Talk to Strangers* (Larman 2005) which perhaps more directly expresses what the rule is about.

In a software context, there is another reason that talking to strangers is a bad idea. In the first pseudo code line, the dean object actually has to know no less than three interfaces: the interface of the department head, the teaching committee head, and the secretary. Consider that some developer changes the signature of method changeSection in the secretary interface: change the parameter list, change the method name, or similar. This would then require a rewrite of the code line in the dean class and regression testing after the change. The same argument applies for changes in the interface for committee heads. Thus the classes become highly coupled. In the second pseudo code line there are only a coupling between two classes, a change in the secretary interfaces does not ripple up into the dean class. Coupling is much lower.

The law can be restated in more operational terms by defining which objects are not strangers.

In a method, you should only invoke methods on

- this
- a parameter of the method
- an attribute of this
- an element in a collection which is an attribute of this
- an object created within the method

While formulated as a "law", it is better considered a good design rule that may be broken if good reasons exist. And one good reason that has become common is **fluent interfaces**, also known as *fluent APIs* (?). Fluent interfaces relies on method chaining to provide a compact and expressive way of defining complex behaviors. The Java stream api, introduced in Java 8, is an excellent example.

Consider a simple class, holding a Danish numerical grade given to a student at an exam (using public instance variables to keep the code short):

```
class StudentExam {
  public int grade;
  public String studentId;
}
```

and next consider a List<StudentExam> studentList collection of a set of given grades. Then I can compute the average score using the streaming api like this:

```
System.out.println("Average grade is: " +
                    studentList
                      .stream()
                      .mapToInt(exam -> exam.grade)
                      .average()
                      .orElse(0.0));
```

This is a short and expressive way of iterating through all elements, select the 'grade' field, and compute the average (or return 0.0 in case a null value is returned).

☞ Rewrite the above average computation into "classic Java" using a for loop and compare the size of the code.

This fluent interface clearly violates the law of Demeter—there are four levels of indirect objects in this single line, and each method call returns an object with its own interface that you have to know.

The tradeoff here is to ensure the interfaces are stable and mature in each indirect object, as is indeed the case with the Java stream API. If the interfaces are mature and never change, the benefits of expressiveness is obvious.

10.5 Summary of Key Concepts

Coupling and cohesion are two qualities that are relatively easy to judge in our code and as they are highly related to maintainability, they are important to consider in practical development. *Coupling* is a measure of the degree of dependency of one software unit to other units. If a unit has many dependencies it is termed *high or tight coupling*. If a unit has high coupling then it is less maintainable as there are many sources that can force a change to the unit. *Cohesion* is a measure of how closely focused the behaviors of a unit is—how well defined its responsibility is. If a unit has a narrow and well defined responsibility it has *high cohesion* and is more maintainable. The reason is that it is easier to locate the relevant unit to modify in case of a defect or requirement update. The *law of Demeter* is a rule of thumb that, if followed, lowers coupling in deep dependency graphs. It states that software code should not collaborate with indirect objects. One notable exception is *fluent interfaces* which uses method chaining between indirect objects to achieve compactness and expressiveness.

10.6 Selected Solutions

Discussion of Exercise 10.1:

The indirect dependencies between units are often the most devilish when it comes to software defects. They are difficult to track down and you are often puzzled when they first appear. And the compiler has no chance of helping you out.

Typical indirect dependencies include: Sharing a common global variable, relying on entries in operating system registries or environment variables, relying on a common file format that units read or write, relying on a specific order of initialization, relying on a specific database schema, relying on specific port numbers or IP addresses, etc. All these create coupling between units in a system but are much less visible couplings than direct object references.

The list is very long and in a development team it can be quite difficult to ensure that everybody understands all couplings between units. The war story in sidebar **??** is a good example of a coupling that was hidden from one team developer. It is also an example of an unfortunate coupling that should have been avoided by the other developers.

Discussion of Exercise 10.2:

Classic examples from the Java libraries are `java.util` that has low cohesion as all sorts of different classes with little or nothing in common are grouped into this package: it contains collection classes, calendar classes, etc. The `javax.swing` package is highly cohesive as only Swing related GUI components are packaged here.

10.7 Review Questions

Define what coupling is and how to achieve weak and tight coupling.

Define what cohesion is and how to achieve low and high cohesion.

Describe the relationship between coupling and cohesion and software reliability and maintainability.

Describe the law of Demeter and its implications. What is its relation to the coupling and cohesion properties?

Describe cases where the law of Demeter should not be adhered to.

10.8 Further Exercises

Exercise 10.3:

Discuss the design patterns you have learnt with respect to coupling and cohesion. Do they increase or decrease coupling? Cohesion? Argue why.

Exercise 10.4:

On page 37 is given a key point: *Keep All Testing Related Code in the Test Tree.* Reformulate this key point using the concepts of coupling and/or cohesion.

Exercise 10.5. Source code directory:
`chapter/refactor/iteration-7`

The final iteration of the pay station system exhibits a complex folder, package, and class structure. Analyze the organization with respect to coupling and cohesion. Consider at least: the contents of the source code and test code tree and the organization and grouping of test cases into separate test classes.