7.3 Introduction to coupling and cohesion

Concept:

The term **coupling** describes the interconnectedness of classes. We strive for loose coupling in a system – that is, a system where each class is largely independent and communicates with other classes via a small, well-defined interface.

Concept:

The term **cohesion** describes how well a unit of code maps to a logical task or entity. In a highly cohesive system each unit of code (method, class, or module) is responsible for a well-defined task or entity. Good class design exhibits a high degree of cohesion.

If we are to justify our assertion that some designs are better than others, then we need to define some terms that will allow us to discuss the issues that we consider to be important in class design. Two terms are central when talking about the quality of a class design: *coupling* and *cohesion*.

The term *coupling* refers to the interconnectedness of classes. We have already discussed in earlier chapters that we aim to design our applications as a set of cooperating classes that communicate via well-defined interfaces. The degree of coupling indicates how tightly these classes are connected. We strive for a low degree of coupling, or *loose coupling*.

The degree of coupling determines how hard it is to make changes in an application. In a tightly coupled class structure, a change in one class can make it necessary to change several other classes as well. This is what we try to avoid, because the effect of making one small change can quickly ripple through a complete application. In addition, finding all the places where changes are necessary and actually making the changes can be difficult and time consuming.

In a loosely coupled system, on the other hand, we can often change one class without making any changes to other classes, and the application will still work. We shall discuss particular examples of tight and loose coupling in this chapter.

The term *cohesion* relates to the number and diversity of tasks for which a single unit of an application is responsible. Cohesion is relevant for units of a single class and an individual method.[1]

Ideally, one unit of code should be responsible for one cohesive task (that is, one task that can be seen as a logical unit). A method should implement one logical operation, and a class should represent one type of entity. The main reason behind the principle of cohesion is reuse: if a method or a class is responsible for only one well-defined thing, then it is much more likely that it can be used again in a different context. A complementary advantage of following this principle is that, when change *is* required to some aspect of an application, we are likely to find all the relevant pieces located in the same unit.

We shall discuss how cohesion influences quality of class design with examples below.

[1] We sometimes also use the term *module* (or *package* in Java) to refer to a multi-class unit. *Cohesion* is relevant at this level, too.