Law of Demeter [Larman98]

This pattern is also known as *Don't Talk to Strangers.*

SYNOPSIS

If two classes have no reason to be directly aware of each other or to be otherwise coupled, then the two classes should not directly interact. Instead of having a class call the methods of another class with which it has no reason to be coupled, you should have it call that method indirectly through another class. Insisting on such indirection keeps a design's overall level of coupling down.

CONTEXT

Suppose you are designing an employee timekeeping system. Your design, so far, includes the classes and associations shown in the class diagram in Figure 4.16. The purpose of the classes in this figure is to determine how many regular and overtime hours an employee worked in a given pay period. Here are descriptions of the classes shown in Figure 4.16:

Employee Instances of the `Employee` class represent an employee.

PayrollRules The rules for paying an employee vary with the laws that apply to the location where the employee works. They may also vary if the employee belongs to a union. Instances of the `PayrollRules` class encapsulate the pay rules that apply to an employee.

PayPeriod Instances of the `PayPeriod` class represent a range of days for which an employee is paid in the same paycheck.

Shift Instances of the `Shift` class represent ranges of time that the employee worked.

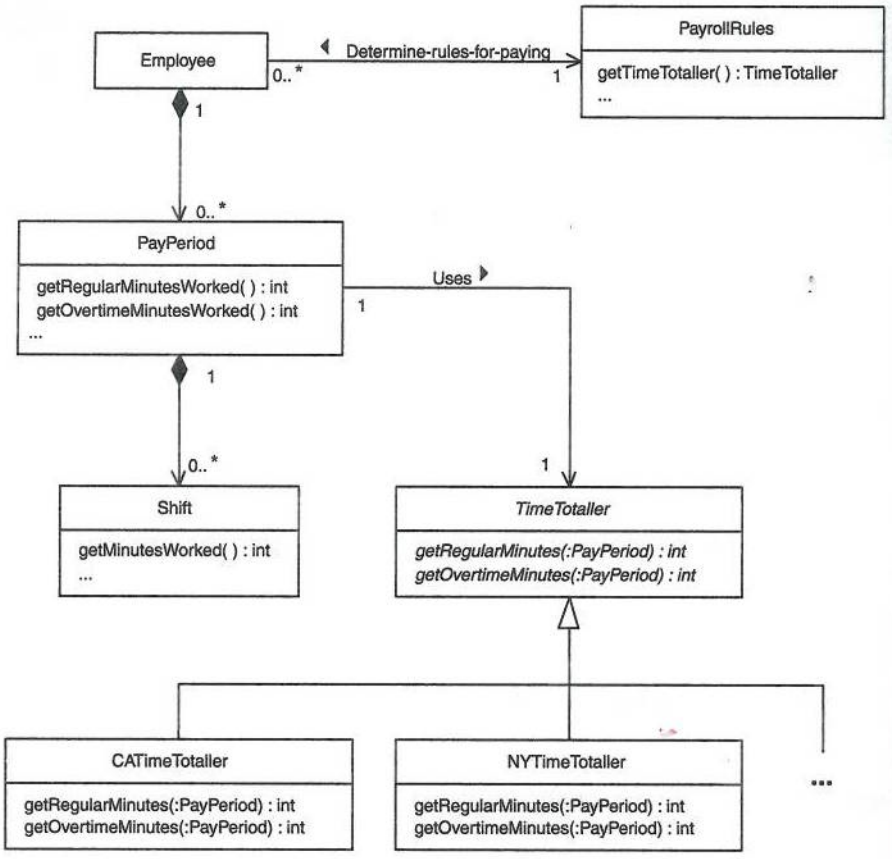

FIGURE 4.16 Time-totalling classes.

TimeTotaller The `TimeTotaller` class is an abstract class that the `PayPeriod` class uses to break the total hours worked during a pay period into regular and overtime minutes.

CATimeTotaller and NYTimeTotaller These are concrete subclasses of `TimeTotaller` that encapsulate the rules for breaking total minutes worked into regular and overtime minutes worked. Each of these classes encapsulates rules for a different locale.

To make these classes carry out their responsibility for computing the number of regular and overtime minutes an employee worked during a pay period, there are some interactions that must occur for which the design does not account:

- The pay period must become associated with an instance of the subclass of `TimeTotaller` appropriate for the employee when the `PayPeriod` object is created.
- The `TimeTotaller` object must be able to examine each shift in the pay period to learn the number of minutes worked in each shift.

The Law of Demeter pattern states that classes that have no reason to be aware of each other should not have any direct interactions. Based on this, the class diagram in Figure 4.17 shows how these interactions should *not* be designed.

The `PayPeriod` class has no reason to know anything about the `PayrollRules` class. The `TimeTotaller` class does have a legitimate reason to be aware of the `Shift` class. However, for a `TimeTotaller` object to directly access the collection of shifts that it needs implies violation of the `Shift` class's encapsulation of how it aggregates collections of shifts. These direct interactions result in a higher level of coupling for the classes that are involved.

These interactions must be indirect in order to occur without creating problems in the design. The collaboration diagram in Figure 4.18 shows the interactions occurring in a way that respects the rest of the design.

The interactions in this diagram are less direct, but they respect the encapsulation of the classes involved and maintain a low level of coupling. The class diagram in Figure 4.19 shows the previous design with methods added to support the interactions of the diagram in Figure 4.18.

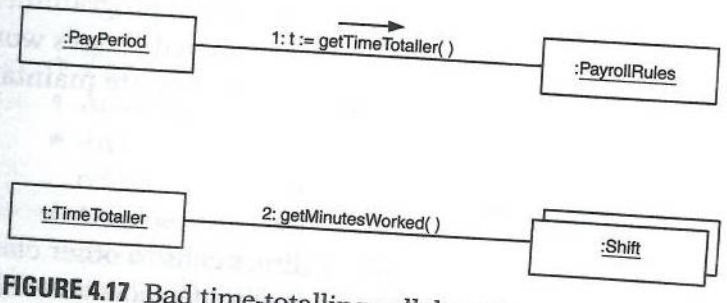

FIGURE 4.17 Bad time-totalling collaboration.

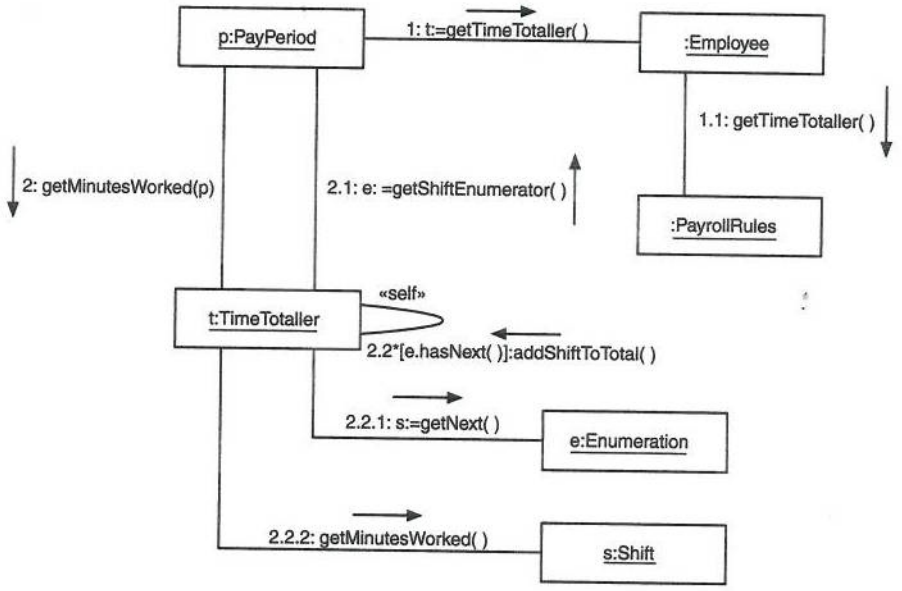

FIGURE 4.18 Good time-totalling collaboration.

FORCES

- Designing a class to directly access all the other classes whose services it needs is the most efficient organization of classes in terms of time spent accessing those services.
- Designing a class to directly access all the other classes whose services it requires can make the class highly coupled to other classes. It also makes the classes more likely to require a change if the structure of the relationships changes. It also makes the design less robust.
- The amount of time lost in making indirect method calls is usually very small. The additional programmer time that is required to make highly coupled classes work correctly and keep them working when they are maintained can be very high.

SOLUTION

Avoid having classes make direct calls to other classes with which they have an indirect relationship. Objects that follow this guide-

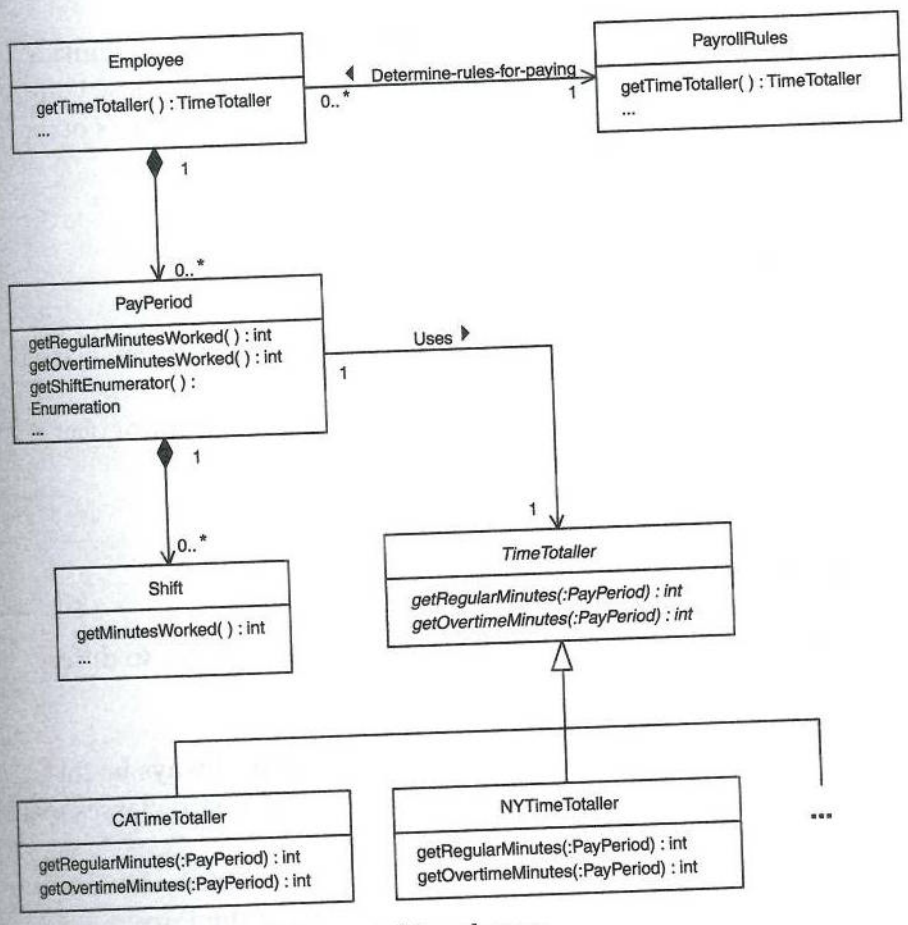

FIGURE 4.19 Enhanced time-totalling classes.

line should make method calls only to objects with which they have one of the following relationships:

- The same object (`this`)
- An object that is passed as a parameter to a method
- An object to which one of its instance variables directly refers
- An object in a collection to which one of its instance variables directly refers
- An object created by the object

Objects that satisfy at least one of these constraints are considered to be *familiars*. Objects that don't satisfy one of these constraints are considered to be *strangers*, hence the pattern's other name—Don't Talk to Strangers.

CONSEQUENCES

- The Law of Demeter pattern keeps coupling between classes low and makes a design more robust.
- The Law of Demeter pattern adds a small amount of overhead in the form of indirect method calls.

IMPLEMENTATION

The guidelines of this pattern apply in varying degrees to different kinds of classes.

- The guidelines in this pattern should almost always be followed for method calls between problem domain classes that correspond to entities in a conceptual model.
- When deciding how to handle calls from problem domain classes to classes added to the design using the Pure Fabrication pattern, there may be other considerations that justify direct calls between the otherwise unrelated classes. The Mediator pattern described in Volume 1 provides an example of this.
- Calls to utility classes that are not specific to the problem domain or the application are generally not subject to the guidelines of this pattern. The same applies to calls made between different architectural layers of a design.

RELATED PATTERNS

Low Coupling/High Cohesion The fundamental motivation for the Law of Demeter pattern is to maintain low coupling.

Pure Fabrication There are sometimes good reasons why calls made to classes added to a design using the Pure Fabrication pattern should violate the guidelines of the Law of Demeter pattern.

Mediator The Mediator pattern (described in Volume 1) provides an example of a class created through pure fabrication that receives direct method calls from classes unrelated to it with a benefit that outweighs the disadvantages of the direct calls.