10.13 Program to an Interface, Not an Implementation[3]

3. Defined in Gamma, Erich et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995, 17–18, and further emphasized and discussed in Bloch, Joshua. *Effective Java*. Upper Saddle River, NJ: Addison-Wesley, 2008.

Implementation inheritance via `extends` has been explained in detail in Chapters 9 and 10. Recall that Java does not allow a class to inherit from more than one superclass.

8

With interface inheritance, a class `implements` an interface describing various `abstract` methods that the new class must provide. The new class also may inherit some method implementations (allowed in interfaces as of Java SE 8), but no instance variables. Recall that Java allows a class to implement multiple interfaces in addition to extending one class. An interface also may extend one or more other interfaces.

10.13.1 Implementation Inheritance Is Best for Small

Numbers of Tightly Coupled Classes

Implementation inheritance is primarily used to declare closely related classes that have many of the same instance variables and method implementations. Every subclass object has the *is-a* relationship with the superclass, so anywhere a superclass object is expected, a subclass object may be provided.

Classes declared with implementation inheritance are tightly coupled—you define the common instance variables and methods once in a superclass, then inherit them into subclasses. Changes to a superclass directly affect all corresponding subclasses. When you use a superclass variable, only a superclass object or one of its subclass objects may be assigned to the variable.

A key disadvantage of implementation inheritance is that the tight coupling among the classes can make it difficult to modify the hierarchy. For example, consider supporting retirement plans in Section 10.5's `Employee` hierarchy. There are many different types of retirement plans (such as 401Ks and IRAs). We might add a `makeRetirementDeposit` method to class `Employee`, then define various subclasses such as `SalariedEmployeeWith401K`, `SalariedEmployeeWithIRA`, `HourlyEmployeeWith401K`, `HourlyEmployeeWithIRA`, etc. Each subclass would override `makeRetirementDeposit` as appropriate for a given employee and retirement-plan type. As you can see, you

quickly wind up with a proliferation of subclasses, making the hierarchy hard to maintain.

As we mentioned in Chapter 9, small inheritance hierarchies under the control of one person tend to be more manageable than large ones maintained by many people. This is true even with the tight coupling associated with implementation inheritance.

10.13.2 Interface Inheritance Is Best for Flexibility

Interface inheritance often requires more work than implementation inheritance, because you must provide implementations of the interface's `abstract` methods—even if those implementations are similar or identical among classes. However, this gives you additional flexibility by eliminating the tight coupling between classes. When you use a variable of an interface type, you can assign it an object of *any* type that implements the interface, either directly or indirectly. This allows you to add new types to your code easily and to replace existing objects with objects of new and improved implementation classes. The discussion of device drivers in the context of `abstract` classes at the end of Section 10.4 is a good example of how interfaces enable systems to be modified easily.

Software Engineering Observation 10.14

Java SE 8's and Java SE 9's `interface` *enhancements (Sections 10.10–10.11)—which allow* `interface`*s to contain* `public` *and* `private` *instance methods and* `static` *methods with implementations—make programming with* `interface`*s appropriate for almost all cases in which you would have used* `abstract` *classes previously. With the exception of fields, you get all the benefits that classes provide, plus classes can implement any number of interfaces but can extend only one class (*`abstract` *or concrete).*

Software Engineering Observation 10.15

Just as superclasses can change, so can interfaces. If the signature of an interface method changes, all corresponding classes would require modification. Experience has shown that interfaces change much less frequently than implementations.

10.13.3 Rethinking the Employee Hierarchy

Let's reconsider Section 10.5's `Employee` hierarchy with composition and an interface. We can say that each type of employee in the hierarchy is an `Employee` that *has a* `CompensationModel`. We can declare `CompensationModel` as an interface with an `abstract` `earnings` method, then declare implementations of `CompensationModel` that specify the various ways in which an `Employee` gets paid:

- A `SalariedCompensationModel` would contain a `weeklySalary` instance variable and would implement method `earnings` to return the `weeklySalary`.
- An `HourlyCompensationModel` would contain `wage` and `hours` instance variables and would implement method `earnings` based on the number of `hours` worked, with `1.5 * wage` for any `hours` over `40`.
- A `CommissionCompensationModel` would contain `grossSales` and `commission-Rate` instance variables and would implement method `earnings` to return `grossSales * commissionRate`.
- A `BasePlusCommissionCompensationModel` would contain instance variables `grossSales`, `commissionRate` and `baseSalary` and would implement `earnings` to return `baseSalary + grossSales * commissionRate`.

Each `Employee` object you create can then be initialized with an object of the appropriate `CompensationModel` implementation. Class `Employee`'s `earnings` method would simply use the class's composed `CompensationModel` instance variable to call the `earnings` method of the corresponding `CompensationModel` object.

Flexibility if Compensation Models Change

Declaring the `CompensationModel`s as separate classes that implement the same interface provides flexibility for future changes. Let's assume `Employee`s who are paid purely by commission based on gross sales should get an extra 10% commission, but those who have a base salary should not. In Section 10.5's `Employee` hierarchy, making this change to class `CommissionEmployee`'s `earnings` method (Fig. 10.7) directly affects how `BasePlusCommissionEmployee`s are paid, because `BasePlusCommissionEmployee`'s `earnings` method calls `CommissionEmployee`'s `earnings` method. However, changing the `CommissionCompensationModel`'s `earnings` implementation does not affect class `BasePlusCommissionCompensationModel`, because these classes are not tightly coupled by inheritance.

Flexibility if `Employees` Are Promoted

Interface-based composition is more flexible than Section 10.5's class hierarchy if an `Employee` gets promoted. Class `Employee` can provide a `setCompensationModel` method that receives a `CompensationModel` and assigns it

to the `Employee`'s composed `Compensation-Model` variable. When an `Employee` gets promoted, you'd simply call `setCompensation-Model` to replace the `Employee`'s existing `CompensationModel` object with an appropriate new one. To promote an employee using Section 10.5's `Employee` hierarchy, you'd need to change the employee's type by creating a new object of the appropriate class and moving data from the old object into the new one. Exercise 10.17 asks you to reimplement Exercise 9.16 using interface `CompensationModel` as described in this section.

Flexibility if `Employees` Acquire New Capabilities

Using composition and interfaces also is more flexible than Section 10.5's class hierarchy for enhancing class `Employee`. Let's assume we decide to support retirement plans (such as 401Ks and IRAs). We could say that every `Employee` *has a* `RetirementPlan` and define interface `RetirementPlan` with a `makeRetirementDeposit` method. Then, we can provide appropriate implementations for various retirement-plan types.