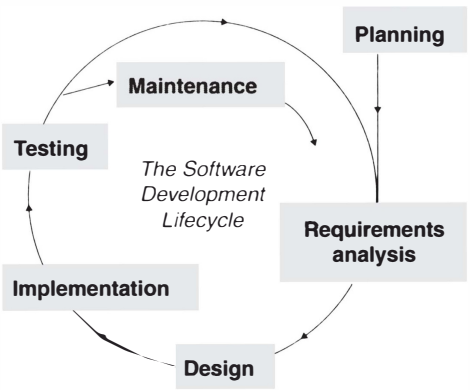
16

The Unified Modeling Language



- What is the UML notation for classes and packages of classes?
- How does UML represent class relationships such as inheritance, aggregation, and dependency?
- What is a sequence diagram?
- How do UML activity diagrams relate to flow charts?

Figure 16.1 The context and learning goals for this chapter

Recall that a "software design" is a representation, or model, of the software to be built. For many years, software engineers relied on a miscellany of somewhat unrelated graphical means for getting across the point of their designs. Their principal means were data flow diagrams, flowcharts, and ways to picture the location of physical files. These were never very satisfactory. With the advent and wide acceptance of object-oriented methods, leaders in the software engineering community began to pool and relate their ideas for notation and graphical representation of software designs. Classes, for example, now needed to be represented in design figures. The Unified Modeling Language (UML) is the result.

In this chapter we introduce the UML, which is now a widely accepted, largely graphical notation for expressing object-oriented designs. The UML standard is managed by the nonprofit Object Management

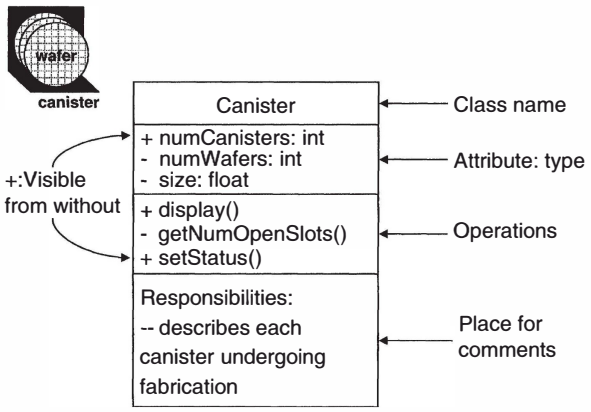


Figure 16.2 Class details in UML

Group consortium of companies (www.omg.org), and takes hundreds of pages to formally specify. As a result, UML tends to be inclusive (some say too much so), incorporating data flow and, in effect, flowcharts; but it contains much more that we do not have space in this book to include. The parts that we do cover, however, are adequate for most of our needs. UML is an excellent step in the direction of improving software engineering, but will probably be improved upon as the discipline evolves.

The chapter describes class relationships in UML, including inheritance (Class Models, Section 16.2) a way to represent control among functions (Sequence Diagrams, Section 16.5), diagrams of state, events, and transitions (Section 16.6), its modernization of flow charts (Activity Diagrams, Section 16.7), and finally data flow diagrams (Section 16.8). Section 16.9 shows an example that combines these.

16.1 CLASSES IN UML

Classes in UML are represented by rectangles containing the class name at a minimum. The detailed version includes attribute and operation names, signatures, visibility, return types, and so on. Figure 16.2 shows a class from the detailed design of an application that controls the flow in a chip manufacturing plant of canisters holding wafers.

Not all of the attributes need be specified in the class model. We show as much detail as needed, neither more nor less. Showing more detail clutters a diagram and can make it harder to understand. Some required attributes may be left to the discretion of the implementers. It is also common to omit accessor functions from class models (e.g., *getSize()* and *setSize()*) since these can be inferred from the presence of the corresponding attributes (*size*).

As an example of a class, consider an application that assists the user in drawing a simple figure of a person using geometric shapes. We'd probably want a *Rectangle* class for this with attributes such as *length* and *breadth*. Since the application is supposed to be smart, we'd want it to use the concept of a foot (e.g., to know where feet belong), so we'd probably want a *Foot* class. By introducing these classes, we are improving the cohesion of related parts, such as the attributes of a foot, the understandability of the design, and its modularity. We are also hiding information until it's needed. This example will be explored as we go through this chapter, and will be described as a whole in Section 16.9.

16.2 CLASS RELATIONSHIPS IN UML

This section discusses the way in which UML collects classes and such in packages. It also introduces relationships called associations.

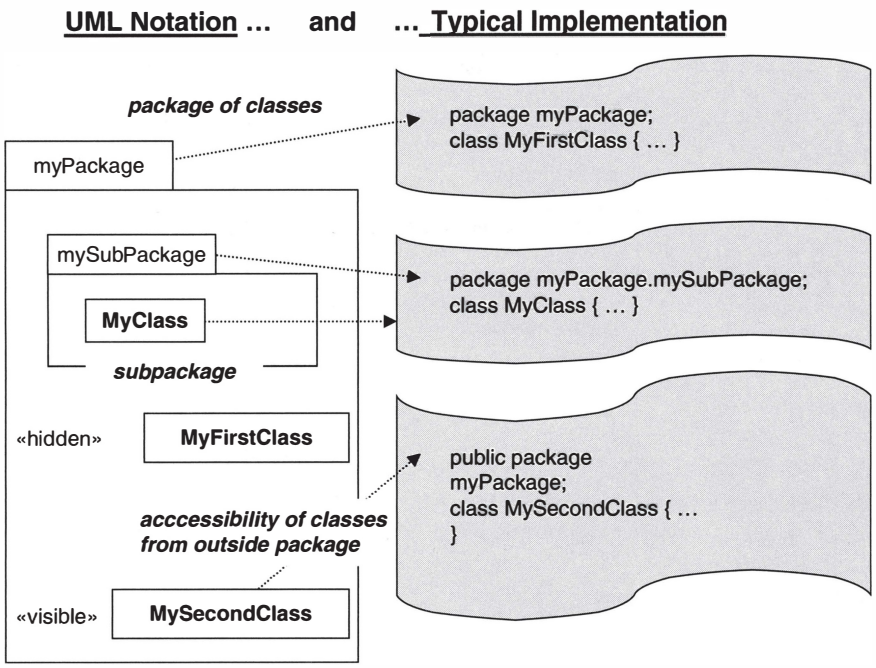


Figure 16.3 UML notation for packages and Java implementation

16.2.1 Packages

The Unified Modeling Language uses the term *package* for collecting design elements such as classes. UML packages can contain subpackages, or any materials associated with an application, including source code, designs, documentation, and so on. Figure 16.3 shows the UML notation for packages and subpackages. Classes within a package can be specified as accessible or as inaccessible from code external to the package. "Package" also happens to be the name of collections of Java classes. Java packages translate into file directories; their subpackages decompose into subdirectories, and so on. The Java implementation of this is shown in Figure 16.3.

16.2.2 Associations

Attributes are one way of showing the properties of a class, and are generally used to denote simple types such as integers or Booleans. *Association* is an additional method of indicating class properties, and is commonly used to denote that objects of two classes depend on each other in a structural way. Associations are drawn with a solid line between two classes. We can annotate the relationship, which may be one- or two-way. Two-way associations are problematical because we need to be sure that both ends of the implied information are kept consistent. This is illustrated in Figure 16.4.

Consider an application that assists the user in drawing a simple figure of a person. We may want a *FootShape* class if the shape of a foot needs to be described (promoting the "sufficiency" of the design). There would be an association between *FootShape* and *Foot*, indicating coupling between the two. This example is described as a whole in Section 16.9.

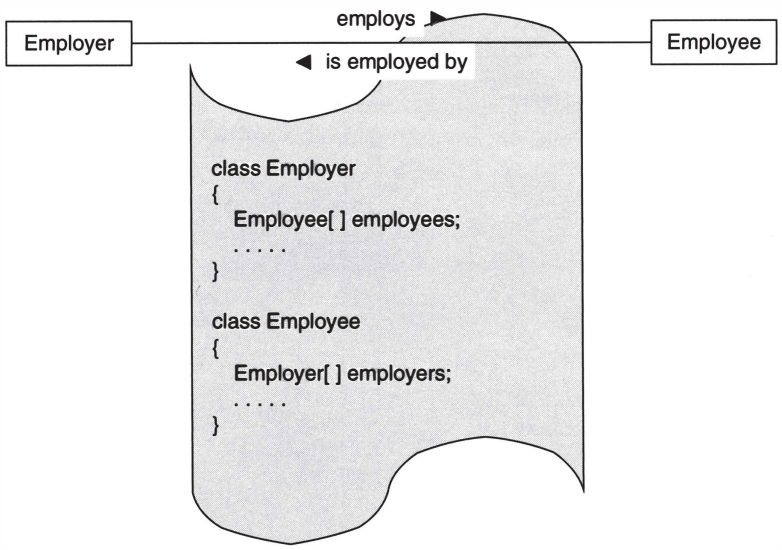


Figure 16.4 UML notation for associations

16.3 MULTIPLICITY

The ends of an association line can be annotated with *multiplicity*, which describes the numerical relationship, or the number of instances, of each class. For example, consider the *Employer/Employee* relationship as shown in Figure 16.5. The "1..3" next to *Employer* indicates that each instance (object) of an *Employee* can be associated with 1–3 instances of an *Employer*. Conversely, the "1..*" next to *Employee* means that each instance of an *Employer* can be associated with a minimum of one *Employer* (with no maximum). In other words, the multiplicity next to a class indicates how many instances of that class are associated with one instance of the class at the other end of the association line. A single value can be used to indicate an exact number of objects. If a multiplicity is not present next to a class, the assumed value is 1. This is also illustrated in Figure 16.5, which shows that one *Employee* is associated with exactly one *PersonalInfo* instance, and vice versa.

16.4 INHERITANCE

In UML, *inheritance* describes a relationship between classes in which one class assumes the attributes and operations of another. It is often thought of as an "is-a" relationship. For example, since an *Employee "is-a" Person*, we can express their relationship with inheritance by saying that an Employee inherits from a *Person*. UML

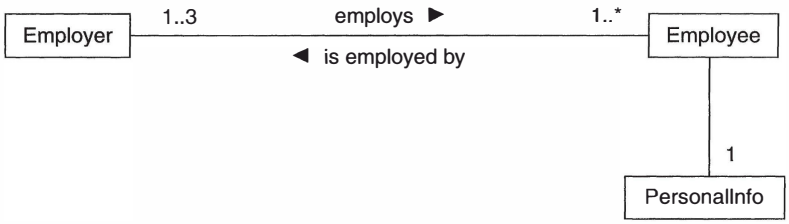


Figure 16.5 Multiplicity of associations in UML

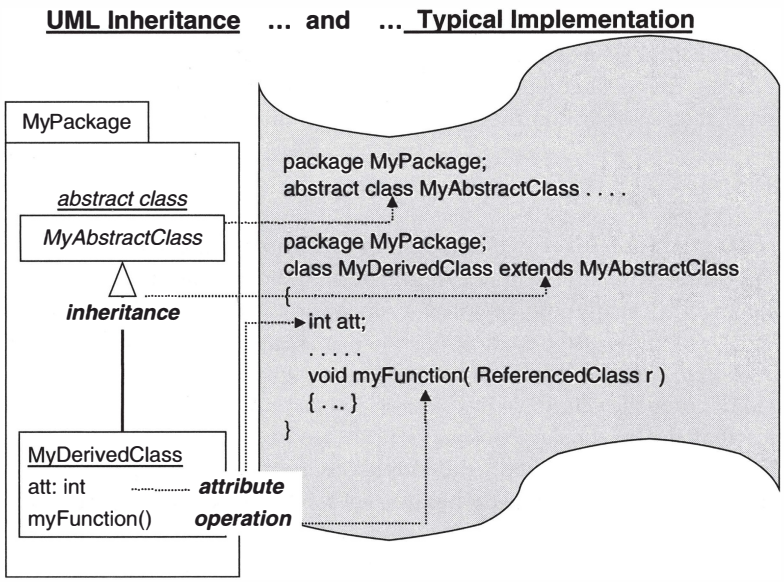


Figure 16.6 Inheritance in UML and Java implementation

indicates inheritance with an open triangle. We refer to the class being inherited from (e.g., *Person*) as a *base* class; the class doing the inheriting (e.g., *Employee*) is a *derived* class.

Figure 16.6 shows an example of a package consisting of two classes: *MyAbstractClass* and *MyDerivedClass*, with *MyDerivedClass* inheriting from *MyAbstractClass*.

Abstract classes—that is, those classes that cannot be instantiated into objects—are denoted with italics. *Interfaces* are collections of method prototypes (name, parameter types, return types, and exceptions thrown). Classes *realize* interfaces by implementing the methods that the interface promises. The UML notation is shown in Figure 16.7.

It is customary to arrange class models so that base classes are physically above derived classes. However, this positioning is not necessary in any technical sense.

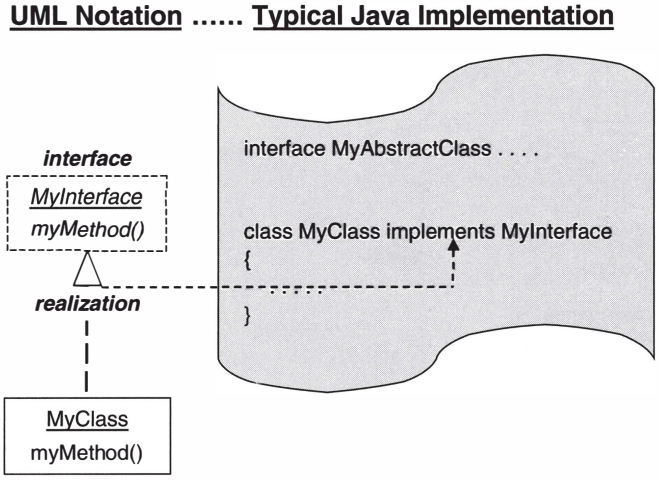


Figure 16.7 Interfaces in UML and Java implementation

Consider again the application that assists the user in drawing a simple figure of a person. Since it deals with various geometric shapes, we may introduce a *GeometricShape* class from which *Rectangle* and such inherit. This saves us from having to repeat code common to *Rectangle*, *Circle*, *Triangle*, and so on, thereby promoting modularity, flexibility, and reusability in the design. This example is described as a whole in Section 16.9.

16.4.1 Aggregation

Aggregation is a type of association that can be thought of as a whole–part relationship. It is denoted with an open diamond next to the aggregator (whole). Aggregation indicates the structural inclusion of objects of one class by another, and is usually implemented by means of a class (whole) having an attribute whose type is the included class (part). Aggregation is shown in Figure 16.8, with both *Company* and *EmployeeDirectory* each considered a "whole," and each consisting of multiple *Employees*, the "parts." The label "emp" next to the diamond denotes the reference in *Company* and *Employer* to the aggregated *Employee*. The use of aggregation implies that if a particular *Employee* is both a part of the *Company* and part of the *EmployeeDirectory*, then the two aggregators "share" the Employee instance that is part of both. That is, if "Jane Doe" is an *Employee*, then one instance of her *Employee* record is created, and both Company and EmployeeDirectory reference that instance.

Returning to the application that assists the user in drawing a simple figure of a person, the association between *Foot* and *FootShape* probably turns out, more specifically, to be an aggregation of *FootShape* by *Foot*. This example is described as a whole in Section 16.9.

16.4.2 Composition

Composition is a stronger form of aggregation in which the aggregated object exists only during the lifetime (and for the benefit of) the composing object—no other object may reference it. The composed object is created and destroyed whenever the composing object is created and destroyed. Composition implies that the composed instance is not shared. For example, Figure 16.9 shows that *Employee* instances are structurally part of *Company* and *EmployeeDirectory*. For any given employee such as "Jane Doe," a separate instance of her Employee object is created as part of *Company* and *EmployeeDirectory*.

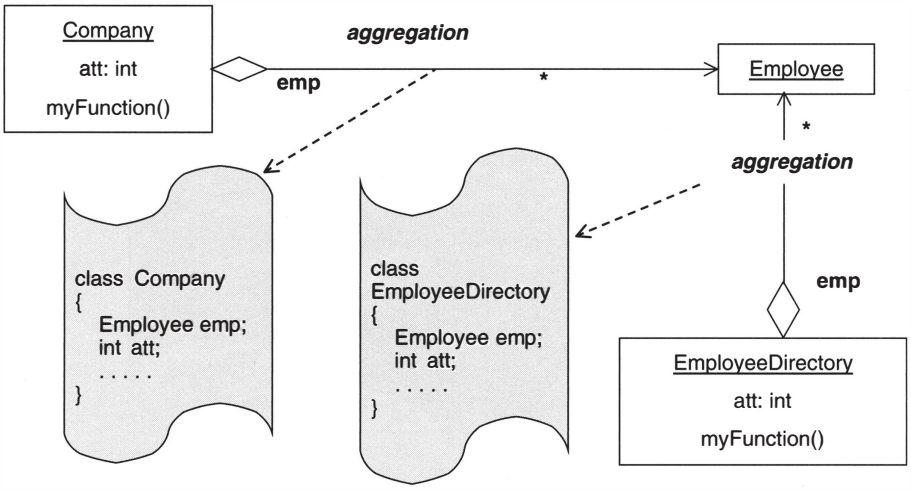


Figure 16.8 UML representation of aggregation and Java implementation

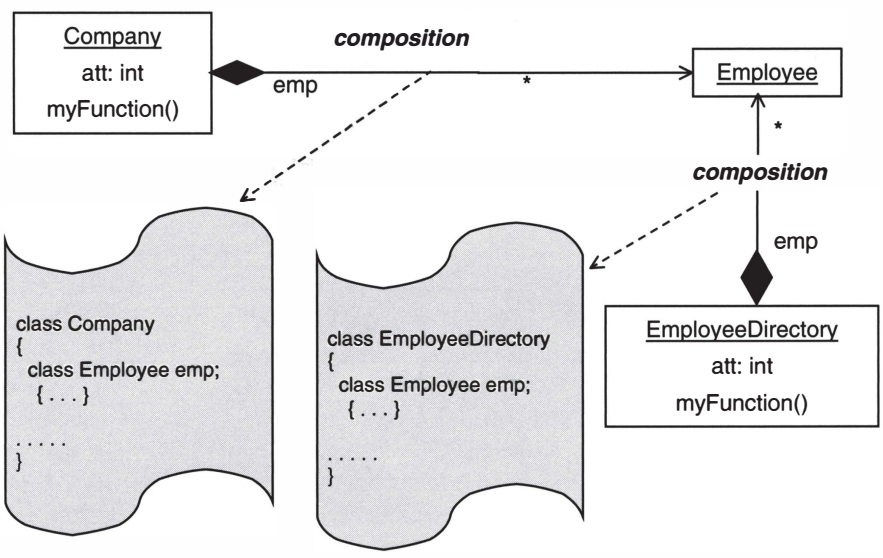


Figure 16.9 UML representation of composition and Java implementation

16.4.3 Dependency

Dependency, denoted with a dotted line arrow, means that one class depends upon another in the sense that if the class at arrow's end were to change, then this would affect the dependent class. Strictly speaking, dependency includes association, aggregation, composition, and inheritance. However, these relationships have their own notation, and we usually reserve dependency to indicate that a method of one class utilizes another class, and to relate packages. This is shown in Figure 16.10.

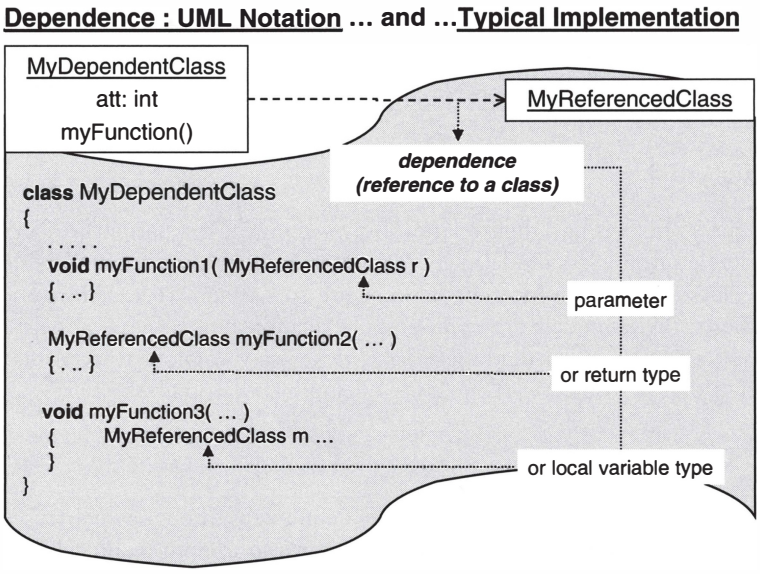


Figure 16.10 UML representation and Java implementation of dependence (excluding inheritance and aggregation)