15 More on Object-Oriented Programming

In this chapter we will survey additional constructs that exist in object-oriented languages. These are not simply clever features added on to the languages, but technical constructs that must be mastered if you wish to become competent in object-oriented programming techniques. The survey is not comprehensive; for details you will need to consult textbooks on the languages.

The chapter is divided into six sections:

1. Structuring classes:
   - Abstract classes are used to create an abstract interface that may be implemented by one or more inheriting classes.
   - Generics (Ada) and templates (C++) can be combined with inheritance to parameterize classes with other classes.
   - Multiple inheritance: A class can be derived from two or more parent classes and inherit data and operations from each one.
2. Access to private components: Are components in the private part of a package or class always private, or is it possible to export them to derived classes or to clients?
3. Class data: This section discusses the creation and use of data components in a class.
4. Eiffel: The Eiffel language was designed to support OOP as the only method of structuring programs; it is instructive to compare the constructs of Eiffel with those of Ada 95 and C++, where support for OOP was added to an existing language.
5. Design considerations: (a) What are the trade-offs between using and inheriting from a class? (b) What can inheritance be used for? (c) What is the relationship between overloading and overriding?
6. We conclude with a summary of methods for dynamic polymorphism.

15.1 Structuring classes

Abstract classes

When a class is derived from a base class, the assumption is that the base class contains most of the required data and operations, while the derived class just adds additional data, and adds or

modifies some operations. In many designs it is better to think of a base class as a framework defining common operations for an entire family of derived classes. For example, a family of classes for I/O or graphics may define common operations such as get and display which will be defined for each derived class. Both Ada 95 and C++ support such abstract classes.

We will demonstrate abstract classes by giving multiple implementations of the same abstraction: the abstract class will define a Set data structure[1] and the derived classes will implement the set in two different ways. In Ada 95, the word abstract denotes an abstract type and abstract subprograms associated with the type:

Ada

```
package Set_Package is
   type Set is abstract tagged null record;
   function Union(S1, S2: Set) return Set is abstract;
   function Intersection(S1, S2: Set) return Set is abstract;
end Set_Package;
```

You cannot declare an object of an abstract type, nor can you call an abstract subprogram. The type serves only as a framework for deriving concrete types, and the subprograms must be overridden with concrete subprograms.

First, we derive a type which implements a set using an array of Booleans:

Ada

```
with Set_Package;
package Bit_Set_Package is
   type Set is new Set_Package.Set with private;
   function Union(S1, S2: Set) return Set;
   function Intersection(S1, S2: Set) return Set;
private
   type Bit_Array is array(1..100) of Boolean;
   type Set is new Set_Package.Set with
      record
         Data: Bit_Array;
      end record;
end Bit_Set_Package;
```

Of course a package body is needed to implement the operations.

The derived type is a concrete type with concrete data components and operations, and it can be used like any other type:

Ada

```
with Bit_Set_Package; use Bit_Set_Package;
procedure Main is
   S1, S2, S3: Set;
begin
```

[1] This is only a small fragment of a useful class for sets.

```
    S1 := Union(S2, S3);
end Main;
```

Suppose now that in another part of the program, you need a different implementation of sets, one that uses linked lists instead of arrays. You can derive an additional concrete type from the abstract type and use it in place of, or in addition to, the previous implementation:

Ada

```
with Set_Package;
package Linked_Set_Package is
    type Set is new Set_Package.Set with private;
    function Union(S1, S2: Set) return Set;
    function Intersection(S1, S2: Set) return Set;
private
    type Node;
    type Pointer is access Node;
    type Set is new Set_Package.Set with
        record
            Head: Pointer;
        end record;
end Linked_Set_Package;
```

The new implementation can be used by another unit; in fact, you can change the implementation used in existing units simply by replacing the context clause:

Ada

```
with Linked_Set_Package; use Linked_Set_Package;
procedure Main is
    S1, S2, S3: Set;
begin
    S1 := Union(S2, S3);
end Main;
```

In C++, an abstract class is created by declaring a *pure* virtual function, denoted by an "initial value" 0 for the function.[2] An abstract class for sets in C++ is:

C++

```
class Set {
public:
    virtual void Union(Set&, Set&) = 0;
    virtual void Intersection(Set&, Set&) = 0;
};
```

It is impossible to define instances of abstract classes; they can only be used as base classes in a derivation:

[2]The syntax was chosen to be similar to the syntax for a null pointer, using "0" rather than a new keyword.

C++

```
class Bit_Set : public Set {
public:
    virtual void Union(Set&, Set&);
    virtual void Intersection(Set&, Set&);
private:
    int data[100];
};

class Linked_Set : public Set {
public:
    virtual void Union(Set&, Set&);
    virtual void Intersection(Set&, Set&);
private:
    int data;
    Set *next;
};
```

The concrete derived classes can be used like any other class:

C++

```
void proc( )
{
    Bit_Set b1, b2, b3;
    Linked_Set l1, l2, l3;

    b1.Union(b2, b3);
    l1.Union(l2, l3);
}
```

Note the difference in syntax between the two languages that comes from the different ways that they approach OOP. In Ada 95 an ordinary function is defined which takes two sets and returns a third. In C++, one of the sets is the distinguished receiver. The intended interpretation of:

```
b1.Union(b2, b3);
```

is that the instance b1, the distinguished receiver of the operation Union, will receive the result of the operation on the two parameters b2 and b3, and use it to replace the current value of its internal data.

For a set class, you may prefer to overload predefined operators like "+" and "*" instead of using names like Union and Intersection. This is possible in both C++ and Ada 95.

All the implementations of the abstract class are in the class-wide type Set'Class. Values of the abstract class-wide type will be dispatched to the correct concrete specific type, that is to the correct implementation. Thus abstract types and operations enable the programmer to write implementation-independent software.

Generics

In Section 10.3 we discussed generic subprograms in Ada, which enable the programmer to create a template for a subprogram and then to instantiate the subprogram for various types. Generics are more commonly applied to Ada packages; for example, a package that maintains a list would be generic on the type of the list element. In addition, it could be generic on a function to compare list elements so that the list can be sorted:

Ada

```
generic
   type Item is private;
   with function "<"(X, Y: in Item) return Boolean;
package List_Package is
   type List is private;
   procedure Put(I: in Item; L: in out List);
   procedure Get(I: out Item; L: in out List);
private
   type List is array(1..100) of Item;
end List_Package;
```

This package can now be instantiated for any element type:[3]

Ada

```
package Integer_List is new List_Package(Integer, Integer."<");
```

The instantiation creates a new type and you can declare and use objects of that type:

Ada

```
Int_List_1, Int_List_2: Integer_List.List;

Integer_List.Put(42, Int_List_1);
Integer_List.Put(59, Int_List_2);
```

Ada has a rich set of generic formal parameter notations which are used in the contract model to restrict the actual parameters to a certain class of types, such as discrete types or floating-point types. In Ada 95, this was generalized to allow generic formal parameters to specify programmer-defined classes of types:

Ada

```
with Set_Package;
generic
   type Set_Class is new Set_Package.Set;
package Set_IO is
   ...
end Set_IO;
```

[3] A generic subprogram formal parameter declared as follows:

```
with function "<"(X, Y: in Item) return Boolean is <>;
```

specifies that if a subprogram with the appropriate signature is visible when instantiation is done, it can be used by default if an actual parameter is not given. This avoids the need to write such things as Integer."<".

This specification means that the generic package can be instantiated with any type derived from the tagged type Set, such as Bit_Set and Linked_Set. All operations from Set such as Union can be used within the generic package, because we know by the contract model that any instantiation will be with a type derived from Set, and hence they will inherit or override these operations.

Templates

In C++, class templates can be defined:

C++

```
template ¡class Item¿
    class List {
        void put(const Item &);
    };
```

Once the class template is defined, you can define objects of this class by supplying the template parameter:

C++

```
List¡int¿ Int_List1;
          // Int_List1 is a List instantiated with int
```

Like Ada, C++ enables the programmer to supply particular subprograms for use in an instantiation by a specific class (a process called *specialization*), or to use default subprograms that exist for the class.

There is an important difference between Ada generics and C++ templates. In Ada, an instantiation of a generic package that defines a type will give you a specific *package* containing a specific type. It takes another step to obtain an object. In C++, the instantiation gives an object directly and there is no specific class defined. To define another object, you just instantiate the template again:

C++

```
List<int> Int_List2;          // Another object
```

The compiler and linker are responsible for keeping track of all instantiations by the same type, to ensure that the code for the operations of the class template is not replicated for each object.

A further difference between the languages is that C++ does not use the contract model, so there is the possibility that an instantiation will cause a compilation error in the template itself (see Section 10.3).

Multiple inheritance

The discussion of derived classes has always been in terms of deriving from a single base, so that a family of classes forms a tree. During the object-oriented design, it is likely that a class will have characteristics of two or more existing classes, and it seems reasonable to derive the class from several base classes. This is called *multiple inheritance*. Figure 15.1 shows that an Airplane can be multiply derived from Winged_Vehicle and Motorized_Vehicle, while the Winged_Vehicle is also a (single) base of Glider. Given the two classes:

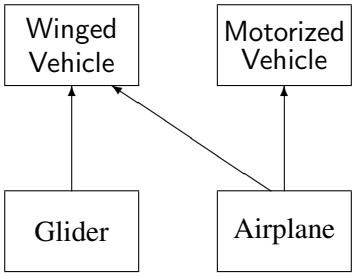

Figure 15.1: Multiple inheritance

C++

```
class Winged_Vehicle {
public:
    void display(int);
protected:
    int Wing_Length;
    int Weight;
};

class Motorized_Vehicle {
public:
    void display(int);
protected:
    int Power;
    int Weight;
};
```

a class can be derived by multiple inheritance:

C++

```
class Airplane :
    public Winged_Vehicle, public Motorized_Vehicle {
public:
    void display_all( );
};
```

The problem that must be solved in order to use multiple inheritance is what to do with data and operations, such as Weight and display, that are inherited from more than one base class. In C++, ambiguities caused by multiply-defined components must be explicitly resolved using the scope resolution operator:

C++

```
void Airplane::display_all( )
{
    Winged_Vehicle::display(Wing_Length);
```

```
    Winged_Vehicle::display(Winged_Vehicle::Weight);
    Motorized_Vehicle::display(Power);
    Motorized_Vehicle::display(Motorized_Vehicle::Weight);
};
```

This is unfortunate since the whole point of inheritance is to allow direct access to unmodified data and operations of the base. The implementation of multiple inheritance is much more difficult than the implementation of single inheritance that we described in Section 14.4. See Sections 10.1c through 10.10c of the Annotated Reference Manual for details.

The importance of multiple inheritance in OOP is the subject of much debate. Some programming languages such as Eiffel encourage the use of multiple inheritance, while others like Ada 95 and Smalltalk have no facilities for multiple inheritance. These languages claim that the problems that multiple inheritance can solve have elegant solutions using other features of the language. For example, we noted above that generic parameters of tagged types in Ada 95 can be used to create new abstractions by combining existing abstractions. Clearly, the availability of multiple inheritance has a deep influence on the design and programming of a system according to object-oriented principles. Thus it is difficult to talk about language-independent object-oriented design; even at the very earliest stages of the design you should have a specific programming language in mind.

15.2 Encapsulation in Java

In Section 13.1, we discussed the fact that C has no encapsulation construct, and in Section 13.5 we noted that the scope resolution operator and the namespace construct in C++ improve on C's coarse approach to global name visibility. For compatibility, C++ does not have an encapsulation construct; instead it depends on "h"-files. Ada has the package construct which supports encapsulation of constructs into modules (see Section 13.3), and the specification and implementation (body) of packages can be separately compiled. with-clauses enable the software engineer to precisely specify dependencies among the modules, and child packages (briefly mentioned in Section 15.3) can be used to develop module structures with hierarchial accessibility.

Java contains an encapsulation construct called a *package*, but confusingly, the construct is closer in spirit to the C++ namespace than to the Ada package! A package is a collection of classes:

Java

```
package Airplane_Package;

    public class Airplane_Data
    {
        int speed;                          // Accessible in package
        private int mach_speed;             // Accessible in class

        public void set_speed(int s) {...}; // Globally accessible
        public int get_speed( ) {...};
```

```
}

public class Airplane_Database
{
    public void new_airplane(Airplane_Data a, int i)
    {
    if (a.speed>1000) {                    // OK !
        a.speed = a.mach_speed;            // Error !
    }

    private Airplane_Data[] database = new Airplane_Data[1000];
}
```

A package can be divided into several files, but a file can contain classes from only one package.[4]

The specifiers public and private are similar to those of C++: public means that the member is accessible anywhere outside the class, while private limits accessibility to other members of the class. If no specifier is given, then the member is visible *within* the package. In the example, we see that the member int speed of class Airplane_Data has no specifier, so a statement within the class Airplane_Database is allowed to access it since the two classes are in the same package. The member mach_speed is declared private, so it is accessible only within the class Airplane_Data in which it is declared.

Similarly, classes have accessibility specifiers. In the example, both classes are declared public, meaning that other packages can access any (public) member of these classes. If a class is declared private, it is only accessible within the package. For example, we might want to declare a private class Airplane_File which would be used within the package to store the database.

Packages are important in Java software development because they allow you to group related classes together, while retaining explicit control over the external interface. The hierarchial library structure simplifies the construction of software development tools.

Comparison with other languages

Java packages serve to control global naming and accessibility in a manner similar to namespaces in C++. Given the declarations in our example, *any* Java program can contain:

Java

```
Airplane_Package.Airplane_Data a;
a.set_speed(100);
```

because the class and method names are public. Without examining the complete source code of a package, you cannot know which classes are imported. There is an import statement which opens up the name space of a package enabling direct visibility. This construct is similar to using in C++ and use in Ada.

[4] You are not required to explicitly declare a package; if you do not, all classes in the file are considered to belong to a common anonymous package.

A key difference between Java and Ada is that in Ada the package specification and the package body are separate. This is not just a convenience for reducing the size of compilations, but a significant factor in managing, developing and maintaining large software systems. The package specification can be frozen, allowing both its implementation and development of other units to proceeding in parallel. In Java, the "interface" of a package is simply the collection of all public declarations. The development of large systems in Java will require software tools to extract package specifications,[5] and to ensure the consistency of the specification and the implementation.

The package construct does give Java one major advantage over C++. The packages themselves use a hierarchial naming convention that enables the compiler and interpreter to automatically locate classes. For example, the standard library contains a function called java.lang.String.toUpperCase. This is interpreted just as an operating system interprets an expanded filename: toUpperCase is a function in package java.lang.String. Java libraries can (but need not) be implemented as hierarchial directories where each function is a file in the directory for its class. Note that the hierarchial names are meaningful only *outside* the language; a subpackage has no special accessibility privileges to its parent. This is in stark contrast to an Ada child package which can access the private declarations of its parent, subject to rules which prevent it from exporting private declarations.

15.3 Access to private components

Friends in C++

Within a C++ class declaration, you can include the declaration of *friend* subprograms or classes, which are subprograms or classes that have full access to private data and operations of the class:

```
class Airplane_Data {
private:
   int speed;

   friend void proc(const Airplane_Data &, int &);
   friend class CL;
};
```

The subprogram proc and the subprograms of the class CL can access private components of Airplane_Data:

```
void proc(const Airplane_Data & a, int & i)
{
   i = a.speed;              // OK, we're friends
}
```

The subprogram proc can then pass on the internal components of the class using reference parameters or pointers as shown. Thus the "friend" has exposed to public view all the secrets of the abstraction.

[5] In Eiffel, this is called the *short* form of a class (Section 15.5).

The motivation behind granting access to otherwise private elements is taken from operating systems which have been designed to explicitly grant specific privileges, called *capabilities*. This concept is less applicable to programming languages because one of the aims of OOP is to create closed, reusable components. From a design point of view, friends are problematic because we are asking the *component* to have knowledge of who is using it, which is certainly not compatible with reusable components that you buy or borrow from another project. Another serious problem with the friend construct is that it may be used too often to "patch" problems in a program instead of rethinking an abstraction. Excessive use of friend will clearly break the abstractions that were so carefully designed.

A valid use of friends is in cases when an abstraction is composed of two distinct elements. In this case, two classes may be declared which are friends of each other. For example, suppose that the class Keyboard needs direct access to the class Display in order to echo a character; conversely, the class Display needs to be able to put a character obtained from a touch-screen interface into the internal buffer of class Keyboard:

```
class Display {
private:
   void echo(char c);
   friend class Keyboard;   // Let Keyboard call echo
};

class Keyboard {
private:
   void put_key(char c);
   friend class Display;     // Let Display call put_key
};
```

The use of friend avoids either of two unsatisfactory solutions: unnecessarily making the subprograms public, or merging the two classes into a single large class just because they have one single operation in common.

Another use of friend is to solve a syntactic problem associated with the fact that a subprogram in a C++ class has a distinguished receiver such as obj1 in the call obj1.proc(obj2). This makes for an asymmetry in subprograms that would otherwise be symmetrical in their parameters. The standard example is overloading of arithmetic operators. Suppose that we wish to overload "+" on complex numbers, while allowing the operation to implicitly convert a floating-point parameter to a complex value:

```
complex operator+(float);
complex operator+(complex);
```

Consider the expression x+y where either x or y can be float and the other complex. The first declaration is correct for complex x and float y because x+y is equivalent to x.operator+(y), which will dispatch on the distinguished receiver of type complex. However, the second declaration for

x+y with x of type float will attempt to dispatch on the type float, but the operator was declared in the class complex.

The solution is to declare all these operators as friends of a class, rather than as operations of the class:

```
friend complex operator+(complex, complex);
friend complex operator+(complex, float);
friend complex operator+(float, complex);
```

While this construct is popular in C++, there is actually a better solution which does not require friend.[6] The operator "+=" can be defined as a member function (see ARM, p. 249) and then "+" can be defined as an ordinary function outside the class:

```
complex operator+(float left, complex right)
{
    complex result = complex(left);
    result += right;          // result is distinguished receiver
    return result;
}
```

Access specifiers in C++

When one class is derived from another, we have to ask if the derived class can access the components of the base class. In the following example, database is declared to be private so it is not accessible in a derived class:

```
class Airplanes {
private:
    Airplane_Data database[100];
};
class Jets : public Airplanes {
    void process_jet(int);
};
void Jets::process_jet(int i)
{
    Airplane_Data d = database[i];   // Error, not accessible !
};
```

If an instance of class Jets is declared, it will contain memory for database but that component is not accessible to any subprogram in the derived class.

There are three access specifiers in C++:

[6] I would like to thank Kevlin A.P. Henney for showing me how to do this.

- A public component is accessible to any user of the class.
- A protected component is accessible within the class, and within a class derived from it.
- A private component is accessible only within the class.

In the example, if database is just protected and not private, it can be accessed from the derived class Jets:

```
class Airplanes {
protected:
   Airplane_Data database[100];
};
class Jets : public Airplanes {
   void process_jet(int);
};
void Jets::process_jet(int i)
{
   Airplane_Data d = database[i];   // OK, in derived class
};
```

though this may not be a good idea because it exposes an abstraction. It would probably be better even for the derived class to manipulate the inherited components using public or protected *subprograms*. Then if the internal representation changes, only a few subprograms need be modified.

C++ allows the accessibility of a class component to be modified during derivation. Normally you would use public derivation (as we have done in all the examples) which retains the base class accessibility. However, you can also derive private'ly in which case both public and protected components become private:

```
class Airplanes {
protected:
   Airplane_Data database[100];
};
class Jets : private Airplanes {       // private derivation
   void process_jet(int);
};
void Jets::process_jet(int i)
{
   Airplane_Data d = database[i];   // Error, not accessible
};
```

Child packages in Ada

In Ada, only a package body has access to private declarations. This makes it impossible to directly share private declarations between packages, in the same way that sharing is possible

with protected declarations in C++. Ada 95 provides for sharing private declarations through an additional structuring facility called *child packages*. We will limit the discussion here to the use of child packages for this purpose, even though they are extremely useful in any situation in which you wish to extend an existing package without modifying or recompiling it.

Given the private type Airplane_Data defined in a package:

```
package Airplane_Package is
   type Airplane_Data is tagged private;
private
   type Airplane_Data is tagged
      record
         ID: String(1..80);
         Speed: Integer range 0..1000;
         Altitude: Integer 0..100;
      end record;
end Airplane_Package;
```

the type can be extended in a child package:

```
package Airplane_Package.SST_Package is
   type SST_Data is tagged private;
   procedure Set_Speed(A: in out SST_Data; I: in Integer);
private
   type SST_Data is new Airplane_Data with
      record
         Mach: Float;
      end record;
end Airplane_Package.SST_Package;
```

Given a package P1 and its child P1.P2, P2 belongs to the scope of the parent P1 as if it were declared immediately after the specification of the parent. Within the private part and the body of the child package, the private declarations of the parent are visible:

```
package body Airplane_Package.SST_Package is
   procedure Set_Speed(A: in out SST_Data; I: in Integer) is
   begin
      A.Speed := I;        -- OK, private field in parent
   end Set_Speed;
end Airplane_Package.SST_Package;
```

Of course the public part of the child package cannot access the private part of the parent, otherwise the child could expose the secrets of the parent package.[7]

[7] A child package can be declared private in which case its visible part *can* access the private part, but then the child package cannot be used outside of the parent and its descendants.

15.4 Class data

Constructors and destructors

A *constructor* is a subprogram that is called when an object of a class is created; similarly, a *destructor* is called when the object is destroyed. Actually, every object (variable) that is defined in any language requires that processing be done when the variable is created and destroyed, if only to allocate and deallocate memory. In object-oriented languages, the programmer is allowed to specify such processing.

Constructors and destructors can be defined in C++ for any class; in fact if you do not define your own, defaults will be supplied by the compiler. The syntax of a constructor is a subprogram with the name of the class, and the syntax of the destructor is the name with a prefixed symbol "~":

C++

```
class Airplanes {
private:
    Airplane_Data database[100];
    int current_airplanes;
public:
    Airplanes(int i = 0) : current_airplanes(i) { };
    ~Airplanes( );
};
```

Upon creation of an Airplanes database, the count of airplanes receives the value of the parameter i, which has a default value of zero:

```
Airplanes a1(15);          // current_airplanes = 15
Airplanes a2;              // current_airplanes = 0
```

When the database is destroyed the code of the destructor (not shown) will be executed.

It is possible to define several constructors which are overloaded on the parameter signatures:

C++

```
class Airplanes {
public:
    Airplanes(int i = 0) : current_airplanes(i) { };
    Airplanes(int i, int j) : current_airplanes(i+j) { };
    ~Airplanes( );
};

Airplanes a3(5,6);         // current_airplanes = 11
```

C++ also has a *copy constructor* which allows programmer-defined processing when an object is initialized with the value of an existing object, or more generally when one object is assigned to another. The full definition of constructors and destructors in C++ is quite complicated; for details see Chapter 12 of the Annotated Reference Manual.

In Ada 95, explicit constructors and destructors usually are not declared. For simple initialization of variables, it suffices to use default values for record fields:

Ada

```
type Airplanes is tagged
   record
      Current_Airplanes: Integer := 0;
   end record;
```

or discriminants (Section 10.5):

```
type Airplanes(Initial: Integer) is tagged
   record
      Current_Airplanes: Integer := Initial;
   end record;
```

Explicit programmer-defined processing is possible by deriving the type from an abstract type called Controlled. This type supplies abstract subprograms for Initialization, Finalization and Adjust (for assignment) that you can override with the specific processing you require. For details, see the package Ada.Finalization described in Section 7.6 of the Ada 95 Language Reference Manual.

Class-wide objects

Memory is allocated for each instance of a class:

C++

```
class C {
   char s[100];
};

C c1, c2;                    // 100 characters for each of c1 and c2
```

Occasionally, it is useful to have a variable that is common to all instances of the class. For example, to assign a serial number to each instance, we would keep a variable last to record the last number assigned. In Ada this is obviously done by including an ordinary variable declaration in the package body:

Ada

```
package body P is
   Last: Integer := 0;
end P;
```

while in C++, a special syntax is needed:

C++

```
class C {
   static int last;          // Declaration
   char s[100];
};

int C::last = 0;             // Definition, accessible outside file
```

The specifier static in this case means that one class-wide object will be allocated. You have to explicitly define the static component outside the class definition. Note that a static class component has external linkage and can be accessed from other files, unlike a static declaration in file scope.

Up- and down-conversion

In Section 14.4 we described how a value of a derived class can be implicitly converted in C++ to a value of its base class. This is called *up-conversion*, because the conversion is upwards from a descendant to any of its ancestors. It is also called *narrowing*, because the derived type is "wide" (since it has extra fields), while the base type is "narrow", having only the fields that are common to all types in its derivation family. Recall that up-conversion occurs only when a value of a derived type is directly assigned to a variable of the base type, not when a pointer is assigned from one variable to another.

Down-conversion from a value of a base type to a value of a derived type is not allowed because we don't know what values to put in the "extra" fields. Consider, however, the case of a pointer to a base type:

C++

```
Base_Class*     Base_Ptr = new Base_Class;
Derived_Class*  Derived_Ptr = new Derived_Class;

if (...) Base_Ptr = Derived_Ptr;
Derived_Ptr = Base_Ptr;    // What type does Base_Ptr point to ?
```

It is certainly *possible* that Base_Ptr will actually point to an object of the derived type; in this case there is no reason to reject the assignment. On the other hand, if the designated object is actually of the base type, we are attempting a down-conversion and the assignment should be rejected. To take care of this case, C++ defines a *dynamic cast* which is conditional on the type of the designated object:

C++

```
Derived_Ptr = dynamic_cast<Derived_Class*>Base_Ptr;
```

If the designated object is in fact of the derived type, the conversion is successful. Otherwise, the null pointer 0 is assigned and the programmer can test for it.

Already in Ada 83, explicit conversion was allowed between any two types derived from each other. This posed no problem because derived types have exactly the same components. It is

possible for them to have different representations (see Section 5.9) but type conversion is perfectly well-defined, because both representations have the same number and type of components.

The extension of derived type conversion to tagged types is immediate in the case of up-conversion from a derived type to a base type. Unneeded fields are truncated:

Ada

```
S: SST_Data;
A: Airplane_Data := Airplane_Data(S);
```

In the other direction, *extension aggregates* are used to supply values to fields that were added during the extension:

Ada

```
S := (A with Mach => 1.7);
```

The fields Speed and so on are taken from the corresponding fields in the value A and the extra field Mach is explicitly given.

When attempting to down-convert a class-wide type to a specific type, a run-time check is made and an exception will be raised if the class-wide object is not of the derived type:

Ada

```
procedure P(C: Airplane_Data'Class) is
   S: SST_Data;
begin
   S := SST_Data(C);      -- What type is C ??
exception
   when Constraint_Error => ...
end P;
```

15.5 The Eiffel programming language

The central characteristics of the Eiffel programming language are:

- Eiffel was built from the ground up as an object-oriented language, rather than by grafting support for OOP onto an existing language.
- In Eiffel, the only way of constructing a program is as a system of classes that are clients of one another or that inherit from one another.
- Since inheritance is the main structuring construct, a standard library of classes (related by inheritance) is central to the language.
- While not part of the "language", a sophisticated programming environment was developed by the Eiffel team. The environment includes language-sensitive support for displaying and modifying the classes, for incremental compilation, and for testing and debugging.

Eiffel departs from Smalltalk (which has similar characteristics) in its insistence on static type checking to go with dynamic polymorphism as in Ada 95 and C++. Eiffel goes further in its attempts to support reliable programming by integrating assertions into the language as we discussed in Section 11.5.

The only program unit in Eiffel is the class: no files as in C and C++, and no packages as in Ada.[8] The terminology of Eiffel is different from that of other languages: subprograms (procedures and functions) are called *routines*, objects (variables and constants) are called *attributes*, and the routines and attributes that comprise a class are called the *features* of the class. Essentially, no distinction is made between functions and constants: like an Ada enumeration literal, a constant is considered simply as a function with no parameters. Eiffel is statically typed like C++ in the sense that assignment statements and parameter passing must have conforming types that can be checked at compile-time. However, the language does not have Ada's wealth of type checking constructs such as subtypes and numeric types.

When a class is declared, a list of features is given:

```
class Airplanes
feature                          -- "public"
   New_Airplane(Airplane_Data): Integer is
      do
         ...
      end; -- New_Airplane
   Get_Airplane(Integer): Airplane_Data is
      do
         ...
      end; -- Get_Airplane
feature { }                      -- "private"
   database: ARRAY[Airplane_Data];
   current_airplanes: Integer;
   find_empty_entry: Integer is
      do
         ...
      end; -- find_empty_entry
end; -- class Airplanes
```

As in C++, a set of features may be grouped and each group can have its accessibility specified differently. A feature-group with a specifier that is the empty set "{ }" exports to no other class, like a private-specifier. A feature-group with no specifier exports to every other class in the system; however, it is unlike a public-specifier in C++ or the public part of an Ada package specification because only read-access is exported. In addition, you may explicitly write a list of classes in the feature-specifier; these classes will be allowed to access the features within the group, like friends in C++.

[8] A disadvantage of restricting the language to this single construct is that the environment must contain an additional tool to specify how a program is created out of a set of classes.

There is no real distinction in Eiffel between predefined types and types defined by the programmer. database is an object of class ARRAY which is predefined in the Eiffel library. Of course, "array" is a very general concept; how are we to indicate the component type of the array? The answer is to use the same method that a programmer would use to parameterize any data type: generics. The predefined class ARRAY has one generic parameter which is used to specify the component type:

```
class ARRAY[G]
```

When an object of type ARRAY is declared,[9] an actual generic parameter must be supplied, in this case Airplane_Data. Unlike Ada and C++, which have a special syntax for declaring predefined composite types, everything in Eiffel is constructed of generic classes using a single set of syntactical and semantical rules.

Generics are widely used in Eiffel, because the library contains definitions of many generic classes that you can specialize for your specific requirements. Generics may also be *constrained* to achieve a contract between the generic class and the instantiation similar to Ada (see Section 10.3). Instead of pattern matching, a constraint is expressed by giving the name of a class from which the actual generic parameter must be derived. For example, the following generic class can be instantiated only by types derived from REAL:

```
class Trigonometry[R -> REAL]
```

You may have noticed that the Eiffel class has no separation between the specification of features and their implementation as executable subprograms. Everything must be in the same class declaration, unlike Ada which divides packages into separately compiled specifications and bodies. Thus the Eiffel language pays for its simplicity by requiring more work from the programming environment. In particular, the language defines a *short* form, in effect the interface, and the environment is responsible for displaying the short form upon request.

Inheritance

Every class defines a type and all classes in the system are arranged in a single hierarchy. There is a class called ANY at the top of the hierarchy. Assignment and equality are defined within ANY but may be overridden within a class. The syntax for inheritance is similar to C++: the inherited classes are listed after the class name. Given the class Airplane_Data:

```
class Airplane_Data
feature
    Set_Speed(I: Integer) is ...
    Get_Speed: Integer is ...
feature { }
    ID: STRING;
```

[9] Note that the declaration does not create the array; this must be done in a separate step which will also specify the size of the array.

```
    Speed: Integer;
    Altitude: Integer;
end; -- class Airplane_Data
```

we can inherit as follows:

```
class SST_Data inherit
    Airplane_Data
        redefine
            Set_Speed, Get_Speed
        end
feature
    Set_Speed(I: Integer) is ...
    Get_Speed: Integer is ...
feature { }
    Mach: Real;
end; -- class SST_Data
```

All the features in the base class are inherited with their export attributes unchanged. However, the programmer of the derived class is free to redefine some or all of the inherited features. The features to be redefined must be explicitly listed in a redefine-clause that follows the inherit-clause. In addition to redefinition, a feature can be simply renamed. Note that an inherited feature can be re-exported from a class even if it was private in the base class (unlike C++ and Ada 95 which do not allow breaking into a previously hidden implementation).

The Eiffel environment can display the *flat* version of a class which shows all the currently valid features even if they were inherited and redeclared somewhere in the hierarchy. This clearly displays the interface of a class and saves the programmer from having to "dig" through the hierarchy to see exactly what was redeclared and what was not.

Eiffel, like C++ but unlike Ada 95, uses the distinguished receiver approach so there is no need for an explicit parameter for the object whose subprogram is being called:

```
A: Airplane_Data;
A.Set_Speed(250);
```

Allocation

Eiffel has no explicit pointers. All objects are implicitly allocated dynamically and accessed by pointers. However, the programmer may choose to declare an object as expanded, in which case it is allocated and accessed without a pointer:

```
database: expanded ARRAY[Airplane_Data];
```

In addition, a class may be declared as expanded and all objects of the class are directly allocated. Needless to say predefined Integer, Character, etc. are expanded types.

Note that given an assignment or equality operator:

```
X := Y;
```

there are four possibilities depending on whether X, Y, neither or both are expanded. In Ada and C++, it is the responsibility of the programmer to distinguish when pointer assignment is intended and when assignment of designated objects is intended. In Eiffel, assignment is transparent to the programmer, and the meaning of each possibility is carefully defined in the language.

The advantage of indirect allocation is that ordinary objects whose type is that of a base class can have values that are of any type whose class is derived from the base type:

```
A: Airplane_Data;
S: SST_Data;

A := S;
```

If the allocation were static, there would be no "room" in the object A to store the extra field Mach of S. Since indirect allocation is used, the assignment is simply a matter of copying the pointer. Compare this with Ada 95 and C++ which require additional concepts: class-wide types and pointers for assignment that preserves the specific type.

Additionally, Eiffel distinguishes between shallow copy and deep copy in assignment statements. Shallow copy just copies the pointers (or the data in the case of expanded objects) while deep copy copies the entire data structures. By overriding the inherited definition of assignment you can choose either meaning for any class.

Dynamic polymorphism follows immediately; given:

```
A.Set_Speed(250);
```

the compiler has no way of knowing if the specific type of the value currently held in A is of A's base type Airplane_Data, or of some type derived from Airplane_Data. Since the subprogram Set_Speed has been redefined in at least one derived class, run-time dispatching must be done. Note that no special syntax or semantics is required; all calls are potentially dynamic, though the compiler will optimize and use static binding where possible.

Abstract classes

Abstract classes in Eiffel are similar to those in C++ and Ada 95. A class or a feature in a class may be declared as deferred. A deferred class must be made concrete by *effecting* all deferred features, that is, by supplying implementations. Note that unlike C++ and Ada 95, you can *declare* an object whose type is deferred; you get a null pointer which cannot be used until a value of an effective derived type is assigned:

```
deferred class Set ...        -- Abstract class

class Bit_Set inherit Set ...-- Concrete class
```

```
S: Set;                 -- Abstract object !
B: Bit_Set;             -- Concrete object

!!B;                    -- Create an instance of B
S := B;                 -- OK, S gets a concrete object
S.Union(...);           -- which can now be used
```

Multiple inheritance

Eiffel supports multiple inheritance:

```
class Winged_Vehicle
feature
    Weight: Integer;
    display is ... end;
end;

class Motorized_Vehicle
feature
    Weight: Integer;
    display is ... end;
end;

class Airplane inherit
    Winged_Vehicle, Motorized_Vehicle
    ...
end;
```

Whenever multiple inheritance is allowed, a language must specify how to resolve ambiguities if a name is inherited from more than one ancestor. Eiffel's rule is basically very simple (even though its formal definition is difficult because it must take account of all the possibilities of an inheritance hierarchy):

> If a feature is inherited from an ancestor class by more than one path, it is shared; otherwise features are replicated.

rename- and redefine-clauses can be used to modify names as necessary. In the example, the class Airplane inherits only one Weight field. Obviously, the intention was for the class to have two Weight fields, one for the airframe and one for the motor. This can be achieved by renaming the two inherited objects:

```
class Airplane inherit
    Winged_Vehicle
        rename Weight as Airframe_Weight;
```

```
    Motorized_Vehicle
        rename Weight as Engine_Weight;
    ...
end;
```

Suppose now that we wish to override the subprogram display. We cannot use redefine since it would be ambiguous which subprogram we are redefining. The solution is to undefine both inherited subprograms and to write a new one:

```
class Airplane inherit
    Winged_Vehicle
        undefine display end;
    Motorized_Vehicle
        undefine display end;
feature
    display is ... end;
end;
```

The Eiffel reference manual discusses in detail the use of rename, redefine and undefine to solve ambiguities in multiple inheritance.

15.6 Design considerations

Inheritance and composition

Inheritance is only one method of structuring that can be used in object-oriented design. A much simpler method is composition which is the inclusion of one abstraction within another. You already know about composition because you know that one record can be included within another:

```
with Airplane_Package;
package SST_Package is
    type SST_Data is private;
private
    type SST_Data is
        record
            A: Airplane_Data;
            Mach: Float;
        end record;
end SST_Package;
```

and in C++, a class can have an instance of another class as an element:

```
class SST_Data {
private:
```

```
    Airplane_Data a;
    float mach;
};
```

Composition is more elementary than inheritance because no new language constructs are required to support it; any support for module encapsulation automatically enables you to compose abstractions. Generics, which in any case are needed in a type-checked language, can also be used to compose abstractions. Inheritance, however, requires sophisticated language support (tagged records in Ada and virtual functions in C++), and run-time overhead for dynamic dispatching.

If you need dynamic dispatching, you must, of course, choose inheritance over composition. However, if dynamic dispatching is not done, the choice between the two is purely a matter of deciding which method produces a "better" design. Recall that C++ requires you to decide when a base class is created if dynamic dispatching is to be done, by declaring one or more subprograms to be virtual; these and only these subprograms will dispatch. In Ada 95, dynamic dispatching will potentially occur on any subprogram declared with a controlling parameter of a tagged type:

```
type T is tagged ...;
procedure Proc(Parm: T);
```

The actual decision whether binding is static or dynamic is made separately for each call. Do not use inheritance when a simple record would suffice.

The basic difference between the two methods is that composition simply uses an existing closed abstraction, while inheritance has knowledge of the implementation of an abstraction. Users of a closed abstraction are protected against modification of the implementation. When inheritance is used, base classes cannot be modified without considering what the modification will do to the derived classes.

On the other hand, every access to a closed abstraction must go through an interface subprogram, while inheritance allows efficient direct access by derived classes. In addition, you may modify an implementation in a derived class, whereas in composition you are limited to using the existing implementation. To put it concisely: it is easy to "buy" and "sell" modules for composition, while inheritance makes you a "partner" of the supplier of a module.

There is no danger in a well-designed use of either method; problems can occur when inheritance is used indiscriminately as this can create too many dependencies among the components of a software system. We leave a detailed discussion of the relative merits of the two concepts to specialized texts on OOP. For the pro-inheritance viewpoint, see Meyer's book (*Object-oriented Software Construction*, Prentice-Hall International, 1988), especially Chapters 14 and 19. Compare this with the pro-composition viewpoint expressed in: J.P. Rosen, "What orientation should Ada objects take?", *Communications of the ACM*, 35(11), 1992, pp. 71–76.

Uses of inheritance

It is convenient to divide the uses of inheritance into several categories:[10]

[10]This classification is due to Ian Muang and Richard Mitchell.

Behaves as An SST behaves as an Airplane. This is a simple use of inheritance for code sharing: operations that are appropriate for Airplane are appropriate for SST, subject to overriding as needed.

Polymorphically substitutable for Linked_Set and Bit_Set are polymorphically substitutable for Set. By deriving from a common ancestor, sets that are implemented differently can be processed using the same operations. Furthermore, you can create heterogeneous data structures based on the ancestor type that contain elements of the entire family of types.

Generically substitutable for Common properties are inherited by several classes. This technique is used in large libraries such as those in Smalltalk or Eiffel, where common properties are factored out into ancestor classes, sometimes called *aspect classes*. For example, a class Comparable would be used to declare relational operations such as "¡", and any class such as Integer or Float that possesses such operations will inherit from Comparable.

Implemented by A class can be created by inheriting its logical functions from one class and its implementation from another. The classic example is a Bounded_Stack which (multiply) inherits its functionality from Stack and its implementation from Array. More generally, a class constructed by multiple inheritance would inherit its functionality from several aspect classes and its implementation from one additional class.

These categories are neither mutually exclusive nor exhaustive; they are intended as a guide to the use of this powerful construct in your software designs.

Overloading and polymorphism

While overloading is a form of polymorphism ("multi-formed"), the two concepts are used for quite different purposes. Overloading is used as a convenience to give the same name to subprograms that operate on entirely distinct types, whereas dynamic polymorphism is used to implement an operation for a family of related types. For example:

C++

```
void proc put(int);
void proc put(float);
```

is overloading because the common name is used just as a convenience and there is no relation between int and float. On the other hand:

C++

```
virtual void set_speed(int);
```

is a single subprogram that happens to be implemented differently for different types of airplanes.

There are technical difficulties in mixing overloading and dynamic polymorphism, and it is recommended that you not combine the use of the two concepts. Do not try to overload within a derived class on a subprogram that appears in the base class:

C++

```
class SST_Data : public Airplane_Data {
public:
   void set_speed(float);    // float rather than int
};
```

The rules of C++ specify that this subprogram neither overloads nor overrides the subprogram in the base class; instead it *hides* the definition in the base class just like an inner scope!

Ada 95 allows overloading and overriding to coexist:

Ada

```
with Airplane_Package; use Airplane_Package;
package SST_Package is
   type SST_Data is new Airplane_Data with ...
   procedure Set_Speed(A: in out SST_Data; I: in Integer);
         -- Overrides primitive subprogram from Airplane_Package
   procedure Set_Speed(A: in out SST_Data; I: in Float);
         -- Overloads, not a primitive subprogram
end SST_Package;
```

Since there is no Set_Speed primitive subprogram with a Float parameter for the parent type, the second declaration is simply an unrelated subprogram that overloads on the same name. Even though this is legal it should be avoided, because the user of the type is likely to be confused. Looking just at SST_Package (and without the comments!), you cannot tell which subprogram overrides and which just overloads:

Ada

```
procedure Proc(A: Airplane_Data'Class) is
begin
   Set_Speed(A, 500);      -- OK, dispatches
   Set_Speed(A, 500.0);    -- Error, cannot dispatch !
end Proc;
```

15.7 Methods of dynamic polymorphism

We conclude this chapter by summarizing dynamic polymorphism in languages for object-oriented programming.

Smalltalk Every invocation of a subprogram requires dynamic dispatching that involves searching up the inheritance hierarchy until the subprogram is found.

Eiffel Every invocation of a subprogram is dynamically dispatched (unless optimized to static binding). Unlike Smalltalk, the possible overridings are known at compile-time, so the dispatching has a fixed overhead based on a jump table.

C++ Subprograms explicitly declared virtual and called indirectly via a pointer or reference are dynamically dispatched. Run-time dispatching has fixed overhead.

Ada 95 Dynamic dispatching is implicitly used on primitive subprograms of a tagged type when an actual parameter is a class-wide type and the formal parameter is of a specific type. Run-time dispatching has fixed overhead.

The language designs differ in the explicit programming and the overhead required for dynamic polymorphism, and these then influence programming style and program efficiency. A clear understanding of the principles involved will help you compare object-oriented languages and improve your ability to design and create good object-oriented programs in whatever language you choose.

15.8 Exercises

1. Implement the Set packages in Ada 95 and the classes in C++.
2. Can an abstract type in Ada 95 or an abstract class in C++ have data components? If so, what might they be used for?

Ada

```
type Item is abstract tagged
   record
      I: Integer;
   end record;
```

3. Write a program for a heterogeneous queue based on an abstract class.
4. Implement the set packages/classes with a generic element type instead of just integer elements.
5. Study in detail multiple inheritance in Eiffel and compare it to multiple inheritance in C++.
6. A standard example of multiple inheritance in Eiffel is that of a fixed-size list implemented by inheriting from both list and array. How would you write such an ADT in Ada 95 which does not have multiple inheritance?
7. What are the dangers of defining protected data in C++? Does this also apply to child packages in Ada 95?
8. Study the structure of the standard library in Ada 95 which makes extensive use of child packages. Compare it with the structure of the standard I/O classes in C++.
9. Study the package Finalization in Ada 95 which can be used to write constructors and destructors. Compare it with the C++ constructs.
10. What is the relationship between assignment statements and constructors/destructors?
11. Give examples of the use of class-wide objects.

12. What is the relationship between the friend specifier in C++ and the package construct in Java.

13. C++ uses the specifier protected to enable visibility of members in derived classes. How does the package construct affect the concept of protected in Java?

14. Compare interface in Java with multiple inheritance in C++.

15. Analyze the differences between namespace in C++ and package in Java, especially with respect to the rules concerning files and nesting.

16. Compare clone and equals in Java with these operations in Eiffel.