6

The Concept of Multiple Inheritance

Multiple Inheritance (MI) is a controversial topic in the object-oriented software domain. Many seasoned OO designers use MI as a way to simplify the class hierarchies in their design. In addition, MI allows more direct modeling of real world situations. For example, you are a result of MI! On the other hand, many designers completely avoid any use of MI in their designs and they argue that MI unnecessarily complicates the design. Further, they emphasize that any design that uses MI can be converted to one that only uses single inheritance. Controversy aside, it is good to know the concept of MI and problem scenarios where it might be applicable. It is essential for the reader to clearly understand that most solutions don't need MI. Single inheritance is one of the more powerful tools very frequently used in good design. Most design solutions don't need MI in their solutions. Many design solutions use MI incorrectly—they use MI where multiple *has-a* relationships would have been more appropriate. Just as with single inheritance, this chapter demonstrates both good and bad usage of MI.

Among the languages discussed in this book, only C++ and Eiffel support multiple inheritance. Smalltalk only supports single inheritance. Hence, the discussions in this chapter exclude Smalltalk.

SIMPLE DEFINITION OF MULTIPLE INHERITANCE

Multiple inheritance is a form of inheritance where a class inherits the structure and behavior of more than one base class. In other words, there are multiple (more than one) parent classes for the child (or derived) class.

Under single inheritance, a derived class inherits from exactly one base class (it has a single parent). The scenario is quite simple, both in terms of implementation and concept. Multiple inheritance, on the other hand, involves multiple base classes. The language (C++ or Eiffel) does not impose any limit on the number of base classes that can be used.[1] If the designer is capable of imbibing the complexity of the MI relationship, the language compiler should not have any problems in processing the code. MI can also be treated as a combination of multiple single inheritance relations, all used at once. The best way to understand the complexity and usage of MI is to use it in an example. And we already have one ready for us from the chapter on single inheritance.

THE UNIVERSITY EXAMPLE

Let's revisit the solution we implemented in the previous chapter. The hierarchy we created is reproduced below in Fig. 6-1.

The class TGraduateStudent was mentioned in the discussions but it was not implemented in the previous chapter. Here is the implementation of the class TGraduateStudent. It is quite simple because a TGraduateStudent has more restrictions on the courses she can enroll for and also more responsibilities (like research and seminars).

```
/ *
  * Need to include the header file for TPerson, TStudent, and TTeacher
    Just for the clarity of this example, here are the enums used earlier
 enum EStudentStatus { eFullTime, ePartTime, eExchange };
```

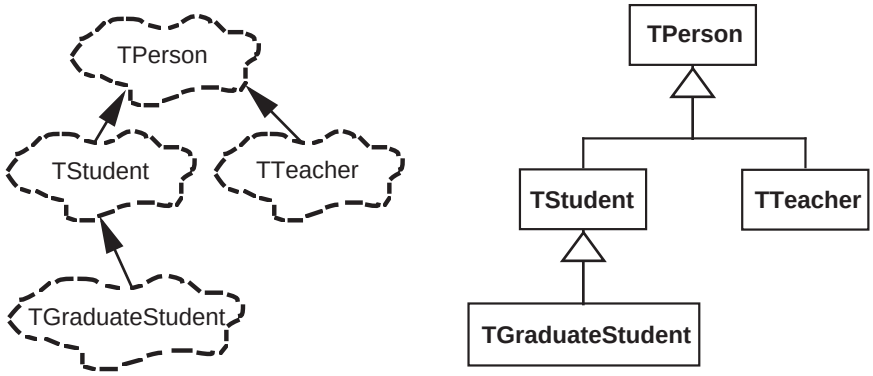


Fig. 6-1

[1]Most design scenarios use two to five base classes, but I have seen some really complicated solutions using 10 to 12 base classes.

```
 enum EDepartment { eAccounting, eBusiness, eEngineering, eMathematics,
     ePhysics, eChemistry, eArts, eUnknown };
And courses are represented by:
    class TCourse {
    public:
         TCourse(const char name[], long id);
         other details not shown
    };
 Array of department names for easy printing
 If you print an enum directly, what you see is an int. But we want to see
 the names of the departments. This array of strings is included for that
     purpose.
static const char *departmentNames[] = {“Accounting”, “Business”,
     “Engineering”,“Physics”, “Arts”, “Chemistry”, “Unknown” };
static const char *statusLabels [] = { “Full time”, “Part Time”,
     “Exchange”, “Unknown” };
*/

/*
 The Graduate student class.
 Very similar to the student class but cannot enroll for low level
 courses (less than GRAD_COURSE_LEVEL). A graduate student always has an
 advisor who guides/controls the academic duties of the graduate student.
*/

// Number of courses a graduate student can enroll for in a semester
const unsigned MAX_COURSES_FOR_GRAD_STUDENT = 5;

 // Cut off course number. A graduate student cannot enroll
 // in courses below this level (the course number).
 const unsigned GRAD_COURSE_LEVEL = 400;

class TGraduateStudent : public TStudent {
     public:
          TGraduateStudent (const char theName[],
                  unsigned long theSSN, // of the graduate student
                  const char theBirthDate[],
                  const char theAddress[],
                  EStudentStatus theStatus,
                  EDepartment theDepartment,
                  const TTeacher& advisorInCharge);

          // Copy Constructor
          TGraduateStudent(const TGraduateStudent& source);
          TGraduateStudent& operator=(const TGraduateStudent& source);
          ~TGraduateStudent();

          // Need to override this method because a graduate student is
          // only allowed enrollment in some predefined courses whereas an
          // undergraduate student can enroll in any course
          bool EnrollForCourse(const TCourse& aCourse);

          void ChangeAdvisor(const TTeacher& newAdvisor);
          TTeacher GetAdvisor() const;
        // Information relevant to a graduate student is printed by
        // this method
```

```
        virtual void Print() const;
        // All other methods are inherited without change
        // because a TGraduateStudent is no different from TStudent
        // with respect to name, address, dropping from a course etc.
        // That's the power of inheritance.
     private:
        // The advisor (a faculty member) for this graduate student
        TTeacher          _advisor;
        // number of courses already enrolled for
        unsigned short    _numCourses;
};

#include <iostream.h>

/* All the arguments for TGraduateStudent are the same as those for
   TStudent except for the advisorInCharge argument. Hence we just call the
   base class (TStudent) constructor and then we initialize the advisor
   object in TGraduateStudent using the copy constructor of TTeacher. We
   receive a fully constructed TTeacher object(advisorInCharge) and we need
   to construct the TTeacher object inside TGraduateStudent using this
   argument. This is the scenario where a new object is being created from
   an existing object. The copy constructor of TTeacher is tailor-made for
   this situation */

TGraduateStudent::TGraduateStudent(const char theName[],
                   unsigned long theSSN,
                   const char theBirthDate[],
                   const char theAddress[],
                   EStudentStatus theStatus,
                   EDepartment theDepartment,
                   const TTeacher& advisorInCharge)
                   // Initialize the base class object first
                : TStudent(theName, theSSN, theBirthDate, theAddress,
                           theStatus, theDepartment),
                // Next initialize the advisor object (uses copy
                // constructor) of TTeacher
                   _advisor(advisorInCharge)
{
      // Now perform the setup for the GraduateStudent part, if any
      _numCourses = 0;
}
// Usual boiler plate code
// The Copy Constructor
TGraduateStudent::TGraduateStudent(const TGraduateStudent& other)
    : TStudent (other), // call base class copy constructor
      _advisor(other._advisor),
      _numCourses(other._numCourses)
{
 // Nothing to do here
}

// The Assignment Operator
TGraduateStudent&
TGraduateStudent::operator=(const TGraduateStudent& other)
{
    // Check assignment to self
    if (this == &other)
```

```
            return *this;

    // Call the base class assignment operator
    TStudent::operator=(other);

    this->_advisor = other._advisor;
    this->_numCourses = other._numCourses;

    return *this;
}

TGraduateStudent::~TGraduateStudent()
{
 // Nothing to do here
}

// Most of the information to be printed resides in the TStudent class. We
// retrieve relevant information from TStudent (and TPerson) and print them
// in the right order. In addition, the name of the advisor is printed
void
TGraduateStudent::Print() const
{
    cout << "Name: " << GetName() << endl;
    cout << "Address: " << GetAddress() << endl;
    EStudentStatus status = GetStatus();
    EDepartment dept = GetDepartment();
    cout << "This person is a " << statusLabels[(int)status]
         << " Graduate Student in the department of "
         << departments[(int) dept] << endl;
    cout << "The Advisor is: " << _advisor.GetName() << endl;
}

bool
TGraduateStudent::EnrollForCourse(const TCourse& aCourse)
{
    // Verify that this grad student hasn't already exceeded the limit
    // for number of courses
    if (_numCourses >= MAX_COURSES_FOR_GRAD_STUDENT) {
          cout << "Graduate Student cannot enroll for more than " <<
               MAX_COURSES_FOR_GRAD_STUDENT << "courses"<< endl;
          return false;
    }

    // Verify that the course level isn't below the set limit for grad
    // students
    if (aCourse.GetCourseId() < GRAD_COURSE_LEVEL) {
        cout << "Sorry, Graduate students cannot enroll for courses below "
             << GRAD_COURSE_LEVEL << "level" << endl;
             return false;
    }

    // Otherwise, ask the TStudent base class to do all the mundane work of
    // enrollment. If enrollment succeeds then increment the # of courses
    // enrolled for.
    bool result = TStudent::EnrollForCourse(aCourse);
    if (result == true)
          _numCourses++;
```

```
    return result;

}
// Return the advisor (a TTeacher) object by value
TTeacher
TGraduateStudent::GetAdvisor() const
{
    return _advisor;
}

// Assign the newAdvisor object to the existing advisor object
void
TGraduateStudent::ChangeAdvisor(const TTeacher& newAdvisor)
{
    _advisor = newAdvisor;
}
```

Code Reuse with Refinement

Pay attention to the simplicity of the TGraduateStudent class. The class interface and implementation appear very simple—don't be fooled by this simplicity; there are some major design features to be understood. Even though a TGraduateStudent also enrolls for courses, most of the work is done in the TStudent class. The TGraduateStudent class does re-implement the EnrollForCourse method but it only checks for certain restrictions that are imposed on graduate students. This is a clear example of a situation where a derived class method accepts most of the behavior of its base class but needs some refinement (or preprocessing/post-processing). To achieve this goal, the derived class method overrides the inherited base class method (EnrollForCourse from TStudent). The implementation of EnrollForCourse in TGraduateStudent does some checks and then invokes the base class method. And after the base class method (TStudent::EnrollForCourse) finishes its part, the TGraduateStudent::EnrollForCourse method does some post-processing based on the outcome of the TStudent::EnrollForCourse. This is another major benefit of inheritance. In the previous chapter, we studied the various benefits of inheritance. This capability to enhance/refine a base class method is one of them. Note that TGraduateStudent does not change the semantics of the behavior of EnrollForCourse—the method is still used for enrolling a student, but more restrictions are enforced. This example illustrates the benefit of code reuse made possible by using inheritance. Definitely, we don't want to rewrite all the uninteresting code for managing the enrollment of a student in several different courses—it has been already done in TStudent. We would like to (re)use it again. But the TGraduateStudent class has more restrictions on enrollments, so we want to enhance (or extend) the code inherited from the TStudent class. And we have successfully done that in TGraduateStudent. This is classic example of code reuse (Fig. 6-2). Many commercial inheritance hierarchies are built this way. One can imagine that each level of inheritance adds a new layer to the system with more capabilities.[2]

[2]Don't be of the impression that code reuse is this easy in all situations—it is not. This example has been chosen to exemplify the concept. Designing a base class for reuse in different scenarios is quite a demanding task.

Inheritance viewed as Layers

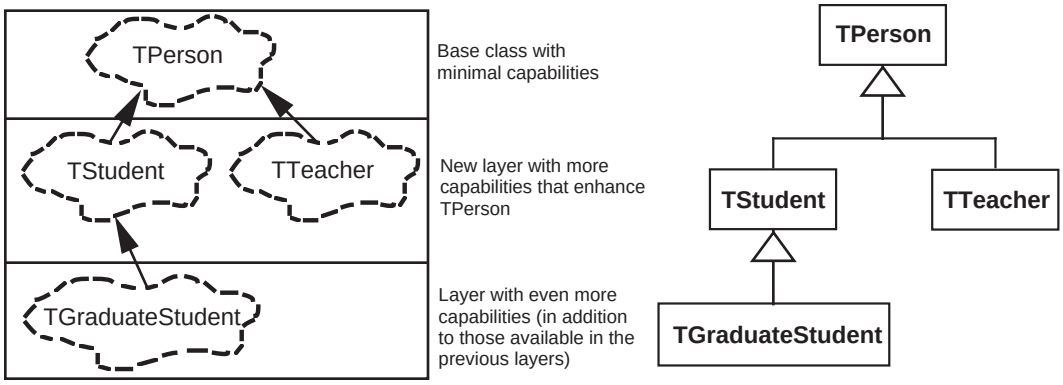


Fig. 6-2

We also want to manage graduate teaching assistants (GTA) in the university. A GTA is a) a graduate student enrolled in the university, and b) teaches some restricted (usually low level undergraduate) courses. Thus, a GTA has a dual role in the university system: Most of the time a GTA is a (graduate) student but at times she behaves like a teacher. We already have students and teachers in our system but not a combination of both. Moreover, the behavior of a GTA is a combination of the behavior of a student and a teacher—and we have already implemented them in our system. There is no fun in implementing the same thing again. We should be able to pull the behavior of TStudent and TTeacher into the class TGradTeachingAsst. Applying our knowledge of single inheritance, this is an extension of the is-a relationship with more than one parent. And that leads us to MI. The new hierarchy is shown in Fig. 6-3.

Class TGradTeachingAsst inherits from both TTeacher and TGraduate-Student. In other words, a TGradTeachingAsst **is-a** TTeacher and **is-a** TGraduate-Student at the same time. A TGradTeachingAsst exhibits the behavior of both TTeacher and TStudent at all times. In the case of single inheritance, we stated that TStudent is-a TPerson always and also that TGraduateStudent is-a TStudent at all times. Now we have extended that definition with TGradTeachingAsst to include two base classes.

One might wonder why this design scenario is considered complex and hard to understand. It seems to be quite simple because we have only scratched the surface of MI. Wait till you see the big picture in the ensuing paragraphs.

The class diagrams show the class relationship among the different classes involved. It is also called a class lattice. (The annotated C++ reference manual uses a slightly different diagram called *directed acyclic graph* (or simply DAG). More precisely a DAG shows the relation between objects under MI. With single inheritance, a DAG would be identical to a class diagram that we have been using. But under MI, the DAG is really the sub-object relationship diagram.)

Now let's see what the syntax looks like in C++.

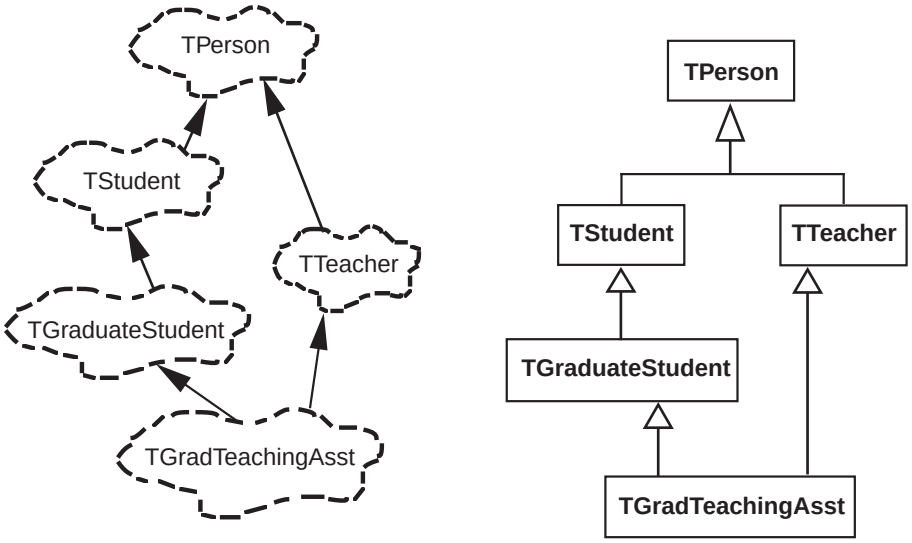


Fig. 6-3

```
class TGradTeachingAsst : public TGraduateStudent, public TTeacher {
     // class member functions and data members
};
```

This code syntax is also an extension of the single inheritance syntax that we are already familiar with. The declaration above says that TGradTeachingAsst has two public base classes. The order of the base class names does not change the effect or meaning of the hierarchy. (The order of base class declarations might be important if you are trying to maintain release to release binary compatibility; see Chapter 13 for details.) We could have also written

```
class TGradTeachingAsst : public TTeacher, public TGraduateStudent  {
     // class member functions/data
};
```

CAUTION The comma (,) character separates the base class names. And don't forget the **public** keyword—it should appear in front of every base class name. If the **public** keyword is omitted from one or more base class names, those base classes become **private** base classes (discussed later in this chapter). Be careful.

THE MEANING OF MULTIPLE INHERITANCE RELATIONSHIPS

Again, let me use the student-teacher hierarchy. Just as single inheritance, MI also asserts the is-a relationship among the derived class and its base classes. In this case, the is-a relationship must be valid between TGradTeachingAsst and TGraduateStudent, as well as between TGradTeachingAsst and TTeacher, at all times. Even though in real life

this student behaves as a teacher at some fixed periods of time and as a graduate student during most other times, that is not the case with MI implementation code. Any TGradTeachingAsst object must be capable of being substituted in place of a TGraduateStudent and TTeacher at all times. There is no relaxation of this rule at any time during the lifetime of a TGradTeachingAsst object. In other words, the *polymorphic substitution principle* must be applicable between TGradTeachingAsst and all other direct and indirect base classes. So we should be able to substitute a TGradTeachingAsst object in place of TPerson, TStudent, TGraduateStudent, and TTeacher without any change in behavior. At any time, a TGradTeachingAsst object is-a TPerson, TStudent, TGraduateStudent, and TTeacher. Furthermore, it has the necessary intelligence to take on the behavior of any of these classes. If the TGradTeachingAsst class (object) is not capable of taking on the behavior of any one of these classes, then rules of MI have been violated and the MI relationship no longer holds. Stated a little differently, MI signals that a derived class has (at least) the behavior of all of its base classes (and probably more) combined together. This should happen naturally in a well designed MI relationship because the derived class automatically inherits the behavior of all of its base classes. The problem is in ensuring the validity of the polymorphic substitution principle under all circumstances.

THE MI SCENARIO

Now let's look at the implementation of the class TGradTeachingAsst.

```
// Need the class header file for TGraduateStudent here
// GraduateTeachingAsst has been added to teacher ranks
enum ERank { eInstructor, eGraduateTeachingAssistant, eAsstProfessor,
                        eAssociateProfessor, eProfessor, eDean };

class TGradTeachingAsst : public TTeacher, public TGraduateStudent {
     public:
          TGradTeachingAsst(const char theName[],
                  unsigned long theSSN, // of the graduate student
                  const char theBirthDate[],
                  const char theAddress[],
                  EStudentStatus theStatus,
                  EDepartment studentDepartment,
                  const TTeacher& advisorInCharge,
                  EDepartment teachingDept);

          TGradTeachingAsst(const TGradTeachingAsst& copy);
          TGradTeachingAsst& operator=(const TGradTeachingAsst& assign);
          ~TGradTeachingAsst();

          // This method intentionally omitted - See text below
          // void Print()const; ❶

          void SetStudentsDepartment(EDepartment dept);
          EDepartment GetStudentsDepartment() const;

          void SetTeachingDepartment(EDepartment dept);
```

```
        EDepartment GetTeachingDepartment() const;
        // and probably other functions
};
```

There are a number of interesting issues to deal with in the class TGradTeachingAsst. Some of the issues are specific to C++ and MI, and some are general complications arising out of the use of MI.

Methods inherited more than once—Name Conflicts: The virtual method Print is inherited by TGradTeachingAsst from TTeacher and from TGraduateStudent. That leads to complications. Here is a piece of code to clarify things.

```
main()
{
    // This is the advisor for the teaching assistant
    TTeacher         Einstein ("Albert Einstein" , 000, "1-1-1879",
                     "Germany/USA", eFullTime, ePhysics);
    // This is the graduate teaching assistant
    TGradTeachingAsst Curie("Marie Curie", 999887777, "06-29-1867",
                "Anytown, France", eFullTime, eChemistry, Einstein,
                 eMatheMatics);
    Curie.Print();
}
```

In this piece of code, when Print() is invoked on the TGradTeachingAsst object Curie, which Print() method should be called? Since Print() is (intentionally) not implemented in TGradTeachingAsst, an inherited Print() method should be called. But TGradTeachingAsst inherits two Print() methods, one from TGraduateStudent and another from TTeacher because of MI. In effect, we are on a road that forks two ways and there is no one at the intersection to guide us correctly.

There is a *name clash* (or name collision) in TGradTeachingAsst with respect to the method Print(). How should one resolve this conflict? Or a better question is: What does the language do to prevent such name clashes?

Resolving Name Conflicts in C++

Under C++, the name conflict would be a compile-time error, and the class TGradTeachingAsst must re-implement the method Print(). Doing so will make the invocation of Print() unambiguous. So when the TGradTeachingAsst class header file is created, it must have the declaration of member function Print() in it.

There is an interesting situation here. When the header file for TGradTeachingAsst is created without the declaration of the method Print() in it (as shown on p. 265), should the compiler detect the missing Print() problem (that we might encounter soon) and flag the missing declaration (of Print) as an error when we try to compile the header file? Or is it better if the compiler detects the error if and when Print() is invoked on an object of TGradTeachingAsst later?

It is better if the compiler detects the name collision as soon as we try to compile the header file of TGradTeachingAsst class. This would be early error detection—the problem is detected at the first possible opportunity and the implementor should fix it immediately. This would prevent more serious problems in the future. Let's call this *conflict detection at the point of declaration* (or early conflict detection). The name collision can

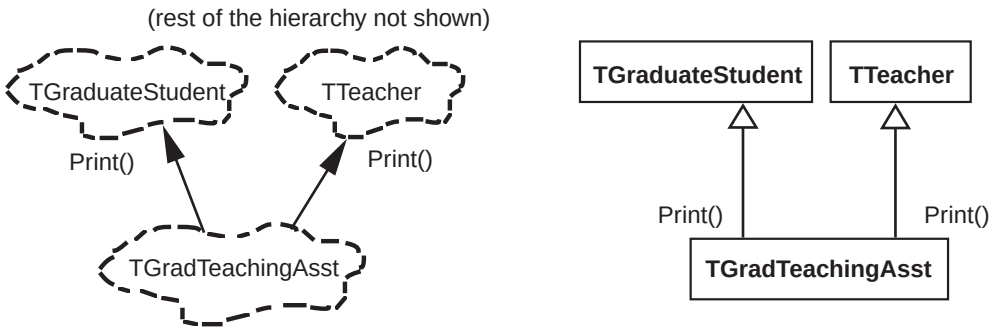


Fig. 6-4

only be detected when the TGradTeachingAsst class declaration file is compiled. This is the earliest time to detect such errors. Once this error is fixed, clients of TGradTeaching-Asst will not have to worry about name collisions when using the TGradTeachingAsst class.

But if the compiler fails to detect the error until some programmer invokes Print() on an object of TGradTeachingAsst it creates new problems for the client. Moreover, the problem is not in the client's code—the error is caused by the missing Print() method in TGradTeachingAsst, and the fix for the problem is in the class TGradTeachingAsst, not in the client's code.[3] Imagine what happens if the TGradTeachingAsst class is implemented and shipped in a library to clients without any testing done on the member function Print(). As soon as some client tries to invoke the Print() method on an object of TGradTeachingAsst class, the compiler would generate an error message (about the ambiguity of Print). And the client cannot fix the error because she does not have source code for the TGradTeachingAsst class. I call this *conflict detection at the point of call* (or late conflict detection) because the error is not detected until someone uses the method. It should be noted that neither TGraduateStudent nor TTeacher can be blamed for the name clash. After all, they are just like any other class. It is the way these two classes are combined by TGradTeachingAsst that causes trouble, and the solution is also to be found there (in TGradTeachingAsst).

Of these two possible error detection schemes, the former (conflict detection at the point of declaration) is much superior because it detects errors at the earliest preventing future problems. But, none of the compilers that I have used provide this level of service. They report the missing Print method as an error only when it is called on a TGradTeachingAsst object. Hopefully, in the years to come better tools (and compilers) would be available that detect these problems (at least report them as a warning).

REMEMBER

This demonstrates the need for thorough testing of classes. In particular, each and every method implemented and inherited by the class must be tested for correct behavior. If this is done diligently, then there wouldn't be any problems for the clients of the class. Also, designers should pay close attention to methods that they inherit from base classes. The proper design in the discussion above is for TGradTeachingAsst to provide an im-

[3]As we shall see later, there is a remedy for such situations (using explicit name qualification) in C++ but it is only a work-around. The real solution is to fix TGradTeachingAsst.

plementation of Print because only TGradTeachingAsst knows what relevant information it should display.

☞ **Pay close attention to member functions that you inherit from base classes and provide an implementation for every method that you inherit from more than one base class.**

C++ Under C++, combining classes that contain members (functions and data members) with the same name (and prototype in case of functions) is not an error. The trouble arises when a programmer tries to use the ambiguous name. If class Z inherits from two base classes X and Y, both of which contain a function f() (virtual or non-virtual), then invoking f() on an object of Z is ambiguous. The programmer must clarify as to which f() is being invoked (X::f() or Y::f()). But, if a programmer just creates an object of Z and never invokes f() on Z, then there is no error. Note that these problems are detected at compile time. One need not have to worry about run-time crashes.

EIFFEL The scenario is much safer under Eiffel. Here, name conflicts are completely prohibited, and the principle of early conflict detection is applied. When a child class inherits the same method from two or more of its parents, then at least one of the inherited methods should be *renamed* to something more appropriate. Of course, both the inherited methods (in our example of TGradTeachingAsst) could be renamed. This scheme ensures that, in a class, any name is always unambiguous. Here is the Eiffel code fragment that shows the renaming concept:

```
class TGradTeachingAsst export
        – all exported features are listed here
     inherit
        TGraduateStudent – name of the inherited class
         rename Print as GSPrint;
        TTeacher   – name of the other inherited class
```

This Eiffel scheme prevents name clash propagation. The disadvantage is that a programmer is forced to invent new names for all clashing feature names. For example, renaming Print inherited from TGraduateStudent as GSPrint is not the best way of doing it because we have to find a new name that clearly reflects the purpose of the method. But finding reasonable names that convey their intent is not an easy task. Note that a renaming in one level of the hierarchy may cause a name clash in a subsequent layer of the hierarchy. Note that the renaming facility is not limited to methods—it is applicable to any feature (data members as well as methods). However, it is more frequently used with methods than with data members.

It is important to remember that name clashes are not limited to member functions. They are possible with data members also, but it is less probable because it is very rare for a derived class to have access to the data members of its base classes. It is very likely that classes have data members with the same name but those data members aren't usually shared with other classes.

Here is the (partially) correct version of TGradTeachingAsst:

```
class TGradTeachingAsst : public TTeacher, public TGraduateStudent {
     public:
```

```
        // other details as before

        // This method must be overridden for proper behavior
        virtual void Print()const;

        void SetStudentsDepartment(EDepartment dept);
        EDepartment GetStudentsDepartment() const;

        void SetTeachingDepartment(EDepartment dept);
        EDepartment GetTeachingDepartment() const;

};
```

And here is a simple implementation of the member function Print.:

```
void TGradTeachingAsst::Print() const
{
     // We get the appropriate pieces of data and print them
     EStudentStatus studentStats = GetStatus(); // no ambiguity here
     EDepartment  studentDepartment = TGraduateStudent::GetDepartment();
     EDepartment  teachingDepartment = TTeacher::GetDepartment();

     // Explicit qualification needed for GetName and
     GetIdentificationNumber
     // Otherwise, it is ambiguous (there are two TPerson objects) in GTA
     cout << "Name: " << TGraduateStudent::GetName() << endl;
     cout << "Id Number: " << TGraduateStudent::GetIdentificationNumber()
                                          << endl;
     cout << "This person is a Graduate Student" << endl;
     cout << " in the department of: " <<
                      departments[(int) studentDepartment] << endl;
     cout << "The graduate teaching assistantship is in the department: "
                    << departments[(int) teachingDepartment] << endl;
}
```

Here is a simple implementation of the constructor:

```
      TGradTeachingAsst::TGradTeachingAsst(const char theName[],
              unsigned long theSSN, // of the graduate student
              const char theBirthDate[],
              const char theAddress[],
              EStudentStatus theStatus,
              EDepartment studentDepartment,
              const TTeacher& advisorInCharge,
              EDepartment teachingDept)
              // Initialize direct base class TGraduateStudent
         : TGraduateStudent(theName, theSSN, theBirthDate, theAddress,
                     theStatus, studentDepartment, advisorInCharge),
              // Initialize direct base class TTeacher
           TTeacher(theName, theSSN, theBirthDate, theAddress,
                     eGraduateTeachingAssistant, teachingDept)
      {
               // Implementation code for TGradTeachingAsst
      }
```

Problem of Ambiguous Base Classes

Another problem with MI is that of ambiguity in implicit derived class to base class conversion (as described in Chapter 5). With single inheritance hierarchies, there are no problems because there is always one base class for any derived class and the implicit conversion happens automatically. But the scenario isn't that simple under MI. Consider this example (in relation to the Teacher-Student hierarchy).

```
// This is an ordinary overloaded function expecting a
// TTeacher argument polymorphically
void foo(const TTeacher& teacher) // ❶
{
     // some code to do processing - not important for this example
}

// This is an overloaded function foo() expecting a
// TGraduateStudent argument polymorphically
void foo(const TGraduateStudent& gradStudent) // ❷
{
     // some code to do processing - not important for this example
}

main()
{
     // This is the advisor for the gta object below
     TTeacher  Super_Smart_Alien("Super_Smart_Alien", 1112223333,
                              "1-1-1900", "Planet Venus", eDean, ePhysics);

     TGradTeachingAsst  gta("Smart_Alien", 777665555, "1-1-2000",
                     "Planet Mars",
                     eFullTime, // Student Status
                     ePhysics,  // Student department
                     Super_Smart_Alien, // Advisor
                     eBusiness // TA department
                     );
     // try calling one of the foo() functions with a TGradTeachingAsst
     // object
     foo(gta); // Which function foo() can be called?
     // Convert gta to TGraduateStudent and call ❷
     // OR convert gta to TTeacher and call ❶
}
```

When we try to invoke foo() with the gta object, there is an ambiguity. Any TGradTeachingAsst class object can be implicitly converted to a TGraduateStudent object or a TTeacher object because TGradTeachingAsst inherits from both of these classes. In other words, TGradTeachingAsst *is-a* TGraduateStudent as well as TTeacher at all times. Both of the foo() functions above are equally good candidates for the call foo(gta). The compiler cannot pick one of the two overloaded foo() functions. Hence, the call is ambiguous. This error is again detected at compile time. This is another potential problem with MI hierarchies. This kind of problem is encountered when programmers add new overloaded functions to existing programs. For example, if one of the foo() functions above were not present, the example above would work without ambigu-

ity. If someone were to add another foo() function later (probably to overcome some other problem), this code (that used to work) will not even compile anymore. Note that foo() could have been a constructor of a class foo, or a set of overloaded member functions in a class.

This problem is very much like the ambiguity of the Print member function discussed earlier. In the case of Print(), the compiler could not resolve between the same member function in two different base classes. In the case of the call to foo(), a TGradTeachingAsst object can be converted to a TTeacher or to a TGraduateStudent. But the compiler cannot pick one conversion over another. Both conversions are valid and equally correct. Here, the ambiguity is because of conversions to accessible base classes.

CAUTION

This problem (of ambiguities) also arises when an existing single inheritance hierarchy is *enhanced* by adding MI. Code that used to compile and run wouldn't even compile. Refrain from using MI until you are aware of the consequences and the caveats.

PRELIMINARY BENEFITS OF MULTIPLE INHERITANCE

It's about time we discussed some of the benefits of using MI. From the example of TGradTeachingAsst above, some benefits are easy to see.

1. We can fully reuse the implementation found in two (or more) different classes and create a new class without much code. This is the case with TGradTeachingAsst. This class gets most of its behavior from the base classes. At the same time, TGradTeachingAsst still upholds the is-a relationship. Stated in a different way, MI allows us to combine classes to create new abstractions, and these new abstractions add more value than the individual abstractions by themselves.

2. MI also helps in modeling relationships that go more than one way. Without MI, a designer is forced to choose among the many possible relationships, even though all of them are valid. When a class exhibits behavior found in more than one (base) class, MI might be appropriate.

3. Multiple inheritance lends itself to a great degree of code reuse. Every additional base class enhances code reuse. Without MI, one would be forced to use the has-a relationship, thereby reducing code reuse (and increasing implementation and testing time).

ALTERNATIVES TO MULTIPLE INHERITANCE

If, for some reason, a project design team decides not to use MI, there are some alternative design scenarios that have been used quite well. The scheme that we are about to explore supports better encapsulation but reduced code reuse. Here is an example.

First Alternative

If we were to implement the TGradTeachingAsst class using *has-a* (aggregation) instead of MI we could write it as follows (just for clarity I am using a different name for this class).[4]

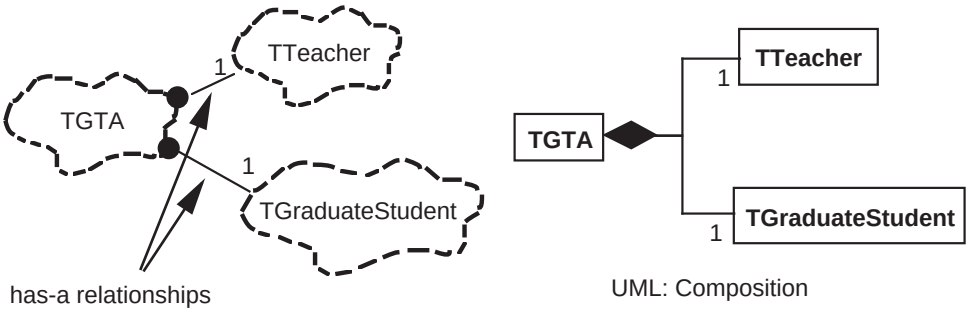
```
// Name of class is different just to avoid confusion

class TGTA {
     private:
          TTeacher _teacherProxy;
          TGraduateStudent _studentProxy;
     public:
          TGTA(const char theName[],
                  unsigned long theSSN, // of the graduate student
                  const char theBirthDate[],
                  const char theAddress[],
                  EStudentStatus theStatus,
                  EDepartment studentDepartment,
                  const TTeacher& advisorInCharge,
                  EDepartment teachingDept);

          TGTA(const TGTA& copy);
          TGTA& operator=(const TGTA & assign);
          ~TGTA();

          void Print()const;

          void SetStudentsDepartment(EDepartment dept);
          EDepartment GetStudentsDepartment() const;

          void SetTeachingDepartment(EDepartment dept);
          EDepartment GetTeachingDepartment() const;
             // and many more methods
};
```

The member functions in the implementation of TGTA have to invoke the corresponding function from one of the proxies (_teacherProxy or _studentProxy) (see Fig. 6-5). For example

```
          void
          TGTA::SetStudentsDepartment(EDepartment dept)
               {_studentProxy.SetDepartment(dept); }
```

This needs to be done for every member function implemented in TGTA because there is no inheritance. Moreover, none of the member functions in TTeacher and TGraduate-Student can be overridden in TGTA, again because there is no inheritance relation. This increases the implementation responsibility for the class TGTA. Class TGTA must provide the member functions needed to interact with TPerson, TStudent, TGraduateStu-

[4]Has-a relationship (aggregation and composition) has been discussed in Chapter 2.



Note: TGraduateStudent still inherits from TStudent which inherits from TPerson. TTeacher also inherits from TPerson. This part of the hierarchy is not shown here.

Fig. 6-5

dent, and TTeacher because clients of TGTA would definitely want access to name, address, courses, etc. of a TGTA. We also have the problem of a duplicated TPerson object—one from TTeacher and another from TGraduateStudent. This problem exists both in the has-a case and also in the MI scenario that we discussed earlier. We shall see a solution for the MI case soon but in the has-a situation there is no easy solution. All these requirements can be satisfied easily with simple implementations, but the implementor must write code for all the required member functions (using the public methods of TGraduateStudent and TTeacher). In case of inheritance, all the methods were inherited automatically and TGradTeachingAsst class had to override only those methods that did not fit its needs. In other words, class TGradTeachingAsst did a selective re-implementation of methods. But in TGTA, almost all the methods that are already implemented in TTeacher and TGraduateStudent that are needed must be re-implemented (using the methods in TTeacher and TGraduateStudent). All this is totally uninteresting code and one would not enjoy doing this implementation of TGTA. Lack of inheritance causes all these problems. This discussion should convince the reader about the power of inheritance.

It might help to look at the conceptual layout of a TGTA object that clearly shows duplication of TPerson objects (Fig. 6-6). In this picture, even though TTeacher inherits from TPerson, we can imagine that TTeacher internally contains a TPerson object.[5] The same argument applies to TGraduateStudent. Every TTeacher object contains all the data members of TPerson, in addition to those it adds. The data members are still private—there is no violation of access rules.

Because of this duplication of TPerson objects, TGTA needs to ensure correct state management when the details of TPerson change due to methods invoked on TGTA. For example, if the address of TGTA needs to be changed, should we change the address stored inside of TPerson within the TTeacher object or the TPerson inside the TStu-

[5]In reality, this is what most C++ compilers do. A derived class object internally contains a base class object. See chapter 13 for details of the C++ object model.

Structure of a TGTA object

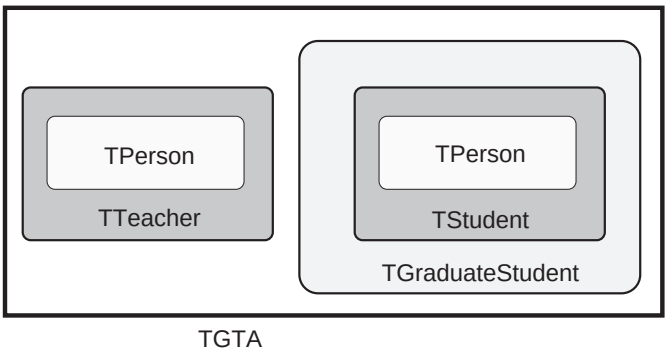


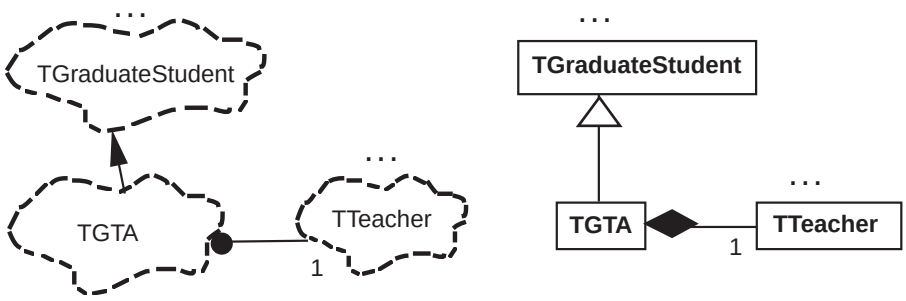
Fig. 6-6

dent object (which is inside TGraduateStudent). If the TGTA implementation completely ignores the TPerson object inside TTeacher and always uses the one inside of TStudent, it might appear that the problem goes away because we never use any data stored in the TPerson object within TTeacher. But what if one of the methods of TTeacher uses the data in its TPerson object through one of the methods in TPerson? There is no way the implementor of TGTA can modify that method in TTeacher to stop this usage. This is one of the problems with duplicated objects. In such an implementation, TGTA object will exhibit different behavior depending on whether TGTA internally invokes the methods in TTeacher or those in TGraduateStudent. Such inconsistent behavior from the same object will confuse the clients and they in turn avoid using such classes. But, there are some benefits in this *has-a* implementation. It provides better encapsulation. A client of TGradTeachingAsst could access the TStudent, TTeacher, TGraduateStudent, and TPerson objects inside of a TGradTeachingAsst object without any restrictions because they are all public base classes of TGradTeachingAsst. There is no such freedom available to a client of TGTA. A client of TGTA can only use the public methods of TGTA and nothing else. The fact that TGTA has an object of TTeacher and TGraduateStudent inside of it is an implementation issue—it is of no use to the client. This provides more flexibility to the implementor of TGTA because she has the freedom to export only those methods that are useful to clients. And she has the freedom to change the internal implementation (for example, using a TStudent object instead of a TGraduateStudent object). Such flexibility is not available with TGradTeachingAsst using MI. This is an important advantage of using has-a relationships and we shall see more advantages of this scheme later in this chapter.

Second Scenario

Let's see if we can use a combination of single inheritance and has-a relationship to our advantage. One approach is to use TGraduateStudent as the base class of TGTA with a has-a relationship with the class TTeacher (Fig. 6-7).

Second Alternative to MI: Combination of is-a and has-a relationships



Note: TGraduateStudent still inherits from TStudent which inherits from TPerson. TTeacher also inherits from TPerson. This part of the hierarchy is not shown here.

Fig. 6-7

The code for TGTA is shown below:

```
class TGTA : public TGraduateStudent {
     private:
          TTeacher  _teacherProxy;
          // Details about the student are in the base class,
          // TGraduateStudent
     public:
          TGTA(const char theName[],
                  unsigned long theSSN, // of the graduate student
                  const char theBirthDate[],
                  const char theAddress[],
                  EStudentStatus theStatus,
                  EDepartment studentDepartment,
                  const TTeacher& advisorInCharge,
                  EDepartment teachingDept);

          TGTA(const TGTA& copy);
          TGTA& operator=(const TGTA& assign);
          ~TGTA();

          virtual void Print()const;

          void SetTeachingDepartment(EDepartment dept);
          EDepartment GetTeachingDepartment() const;
             // and many more methods
};
```

The situation is less complex now because TGTA inherits all the features of TGraduate-Student (indirectly those of TStudent and TPerson also). Therefore, TGTA does not have to re-implement all the methods that treat TGTA as a graduate student. But TGTA must implement (using the corresponding member functions of TTeacher) those methods that add the capability of a teacher to TGTA. But this is much easier than the first scenario, which uses pure has-a relationships. We still have the problem of duplicated TPerson objects—TGTA inherits a TPerson object indirectly from TGraduateStudent and TTeacher inherits from TPerson, but controlling the duplicated object is somewhat easier in this scheme. It is better to use the TPerson inherited through TGraduateStudent

(ignoring the one in TTeacher). Since TGTA controls all access to TTeacher, it might be able to ensure that (with some effort) the TPerson data in TTeacher is not used anywhere.

REPEATED INHERITANCE

The original inheritance hierarchy of TGradTeachingAsst is reproduced here in Fig. 6-8 for easy reference.

So far, we have conveniently ignored the problem of duplicated objects inside of a derived class object (inside TGradTeachingAsst in our example). This is a major problem with multiple inheritance. When the same class X appears as the base class of more than one parent class (i.e., indirect base class), then the completed object of the derived class will have more than one object of X. In Eiffel, it is legal for a class to inherit *directly* from the same base class more than once, but C++ does not allow it. The situation where a class appears as the (direct or indirect) base class more than once in a hierarchy is called *repeated inheritance.* In our example, TPerson is the base class for both TTeacher and TStudent. When TGradTeachingAsst inherits from both TTeacher and TGraduateStudent, it is indirectly inheriting TPerson twice. We end up with a TGradTeachingAsst object with two TPerson objects inside it. What is really duplicated are the data members of the class TPerson. Every TGradTeachingAsst object ends up with two instances of TPerson and each of those TPerson objects have their own data members—_name, _address, _ssn, and _birthDate. Any TGradTeachingAsst is a unique person with a name, address, and such. We don't need two copies of such data. But that's what we get by this default style of MI.

NOTE C++ does not allow *direct repeated inheritance*. For example, the following piece of code is incorrect.

```
class TTeacher : public TPerson, public TPerson { /* ... */ }; // Wrong
```

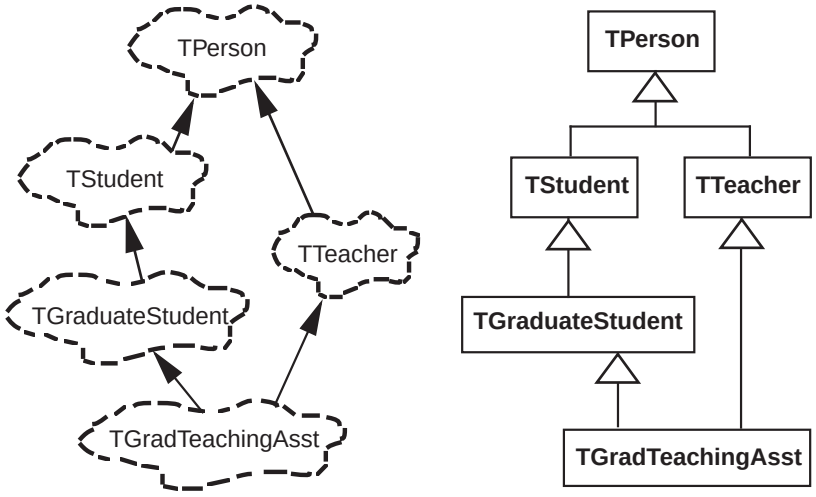


Fig. 6-8

Here, TTeacher inherits directly from TPerson twice. It is impossible to distinguish between the two TPerson objects within TTeacher. But, indirect repeated inheritance (as in TGradTeachingAsst) is perfectly correct.[6]

The problem with TGradTeachingAsst is that TTeacher and TStudent share the structure and behavior of the same class, TPerson, but they don't share the same TPerson object. When a TGradTeachingAsst object is created, it in turn instantiates (calls the constructor of) its base class, TGraduateStudent, which in turn instantiates TStudent, which in turn instantiates a TPerson object with same name, address, etc. In the same TGradTeachingAsst object, when TTeacher is instantiated, it in turn instantiates TPerson with the same name, address, etc. This creates a TGradTeachingAsst object with two TPerson objects (see Fig. 6-9 and code below).

Structure of a TGradTeachingAsst object

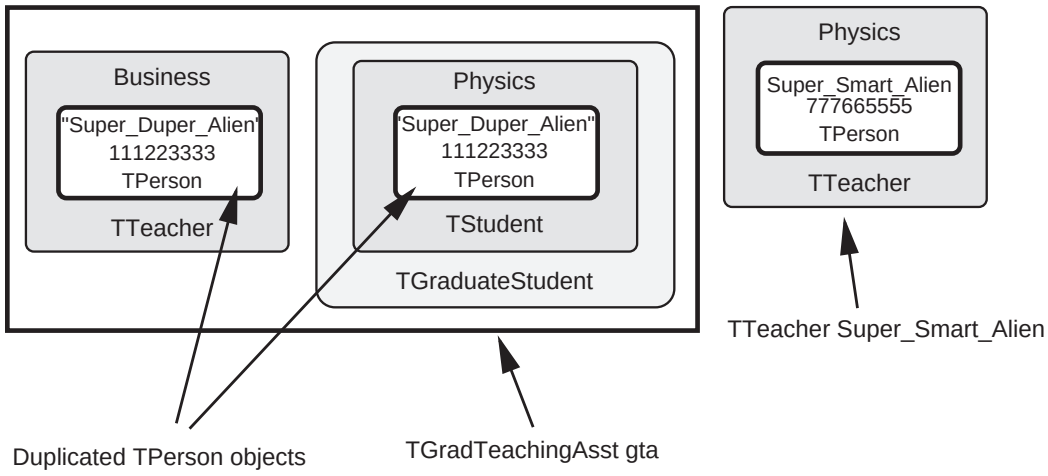


Fig. 6-9

```
main()
{
    // This is the advisor for the gta object below
    TTeacher  Super_Smart_Alien("Super_Smart_Alien", 777665555, "1-1-90",
                                    "Planet Venus", eDean, ePhysics);

    TGradTeachingAsst  gta("Super_Duper_Alien", 1112223333, "1-1-66",
                    "Planet Mars",
                    eFullTime,               // Student Status
                    ePhysics,                // Student department
                    Super_Smart_Alien,       // Advisor
                    eBusiness                // TA department
                    );
}
```

In addition to wasting space, this duplication of TPerson objects causes inconsistent behavior also. For example, if the name of a TGradTeachingAsst is to be changed and if

[6]In Eiffel, direct repeated inheritance is allowed.

the implementation of TGradTeachingAsst ends up changing the _name data member in the TPerson within TGraduateStudent (and not that in TTeacher) then anyone accessing the TPerson object through the TTeacher path would get the old name and clients accessing the TPerson through the TGraduateStudent path will get the new name. Here is a code snippet that demonstrates this situation.

```
main()
{
   // This is the advisor for the gta object below
   TTeacher   mickey("Mickey Mouse", 1112223333, "1-1-1928", "Disney Land",
                                                        eDean, eArts);

   TGradTeachingAsst  bugs("Bugs Bunny", 777665555, "12-1-1940",
                        "Looney Tunes", // address
                        eFullTime,      // Student Status
                        ePhysics,       // Student department
                        mickey,         // Advisor
                        eArts           // TA department
                     );
   // Make a TGraduateStudent pointer point to a TGradTeachingAsst object
   // This is fine because a TGradTeachingAsst is a TGraduateStudent
   TGraduateStudent*  gsp = &bugs
   /*
    Now change the name of the TGradTeachingAsst using the TGraduateStudent
    interface. When we call SetName() using gsp, we are invoking
    SetName as seen in the interface of TGraduateStudent. This will
    access the TPerson which TGraduateStudent inherits from
    TStudent. But nothing has changed in the TPerson inherited by
    TTeacher. It still has the old name "Bug Bunny"
   */
   gsp->SetName("Daffy Duck");
   gsp->Print(); // Print invoked through the TGraduateStudent interface

   /*
     Next create a pointer to TTeacher and access the TGradTeachingAsst
     object through the TTeacher interface. Now we get the old name that
     was stored in TPerson
    */
   TTeacher*  tp = &bugs;
   // The line below will print the old information: "Bugs Bunny"
   cout << "Name through Teacher is: " <<  tp->GetName() << endl;
   // The next line will print the new information: "Daffy Duck"
   cout << "Name through TGraduateStudent is: " << gsp->GetName() << endl;

}
```

In this scenario, for the same object, the result of GetName() is different based on the calling interface. This kind of behavior is highly misleading to clients and they cannot fix the problem easily. The correct solution is to fix the hierarchy such that TPerson is not duplicated. Another problem is that of ambiguity; when a TGradTeachingAsst object needs to be used polymorphically where a TPerson is expected, there is no direct conversion to TPerson possible because there is more than one TPerson object in TGradTeachingAsst. Here is an example:

```
// This is just a simple function accepting a TPerson object
// polymorphically. Can be called with any derived class object of TPerson
void foo(const TPerson& aPerson)
{
     // some code - not important for this example
}

main()
{
     // This is the advisor for the gta object below
     TTeacher  mickey("Mickey Mouse", 1112223333, "1-1-1928",
                              "Disney Land", eDean, eArts);

     TGradTeachingAsst  bugs("Bugs Bunny" 777665555, "12-1-1940,
                     "Looney Tunes",
                     eFullTime, // Student Status
                     ePhysics,  // Student department
                     mickey, // Advisor
                     eArts // TA department
                     );

     // Call foo() polymorphically using the TTeacher object
     foo(mickey); // Works fine because TTeacher is a TPerson

     // Try to do the same with TGradTeachingAsst object
     // Call foo() with the object
     foo(bugs); // Ambiguous conversion to TPerson - fails
}
```

The call to foo() using the bugs object is ambiguous because there isn't a unique TPerson object within a TGradTeachingAsst object (as shown in Fig. 6-9). Hence, the implicit conversion to TPerson is not possible.

What we really need is a facility by which we can share specified data members of the class TPerson in a TGradTeachingAsst object, while at the same time duplicating those that need to be duplicated in hierarchies where there is more than one object of TPerson. In other words, it will be very nice to have control over the sharing of individual data members of the class TPerson. If we could retain only one copy of the data members _name, _ssn, and _birthDate inside of TGradTeachingAsst objects, our problem would be solved. But, remember that private data members are not accessible (nor shareable) outside the class under C++. So it is not possible to have control over the individual data members of the class TPerson. The next best thing we can expect is to keep only one copy of the TPerson object inside of TGradTeachingAsst objects. This is made possible by *virtual base classes*.

SOLUTION FOR REPEATED INHERITANCE

Sharing objects with Virtual Base classes in C++

By making the base class TPerson a virtual base within TGradTeachingAsst, we get only one TPerson object within a TGradTeachingAsst. But notice that the TPerson class is a direct base of TTeacher and TStudent and not of TGradTeachingAsst.

Therefore, TTeacher and TStudent should share the TPerson object when they (TTeacher and TStudent) are combined as part of TGradTeachingAsst objects. Hence, the TTeacher and TStudent class should declare their intentions of sharing the TPerson object. This is done by the following declaration:

```
class TTeacher : virtual public TPerson {
     // all the rest of the code
};

class TStudent : public virtual TPerson {
     // all the rest of the code
};
```

Notice that the ordering of the keywords virtual and public is not important. In the above declaration, TPerson is a **virtual** base for both TTeacher and TStudent. It implies that in any hierarchy where a TTeacher and TStudent are direct or indirect base classes, then the TPerson object would be shared by them (TTeacher and TStudent). In other words, TPerson is not an absolute (or real) base class, as we have seen so far in other inheritance hierarchies. With this change, the real class hierarchy is as shown in Fig. 6-10.

Objects of the class TGradTeachingAsst will have a single object of TPerson inside them (as opposed to two in the previous case). Now, TPerson is referred to as a **virtual base class**. Needless to say, this scenario is not easy to understand without some good explanation of the different possibilities and responsibilities.

1. The TPerson object is shared only within objects of TGradTeachingAsst. If we create a stand alone TTeacher, TStudent, or a TGraduateStudent object, they all get their own TPerson object—nothing is shared among them. This is perfectly sensible.

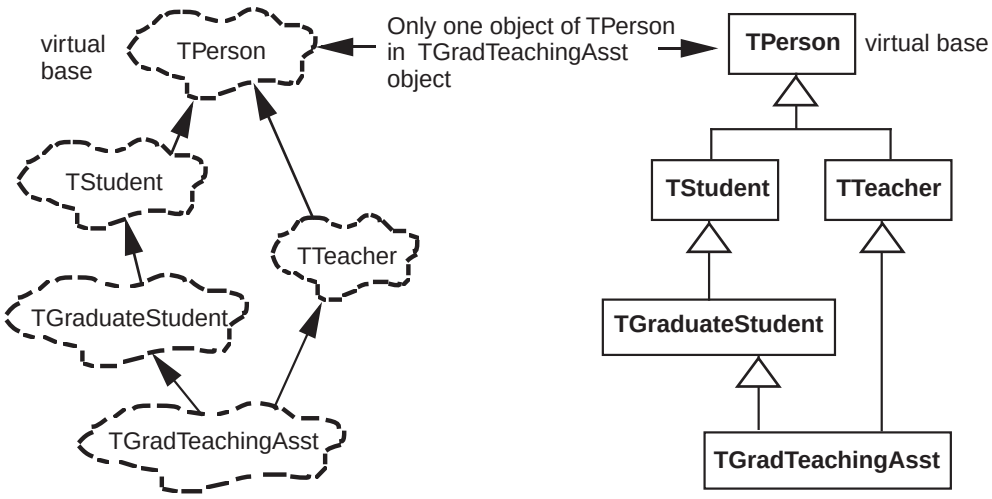


Fig. 6-10

Any TTeacher object has nothing in common with any other TStudent object (unless they are part of the same TGradTeachingAsst object).

2. When a TGradTeachingAsst object is instantiated, the compiler recognizes the virtual base TPerson and only one object of TPerson is created. This ensures that a TGradTeachingAsst has a unique name, address, etc.

3. Even though the TPerson object is shared only when a TGradTeachingAsst object is created, the responsibility and forethought to ensure this lies with the TTeacher and TStudent classes—they have to declare the base class TPerson to be a virtual base. Making such decisions is not a trivial task.

Fig. 6-11 clearly demonstrates the presence of TPerson objects in various scenarios.

In effect, classes TTeacher and TStudent, by making TPerson a virtual base, state that they will share a single TPerson object between them when they are combined in a derived class, by virtue of multiple inheritance. Thus, a virtual base class signals sharing of object(s). In TGradTeachingAsst, all the objects (TGradTeachingAsst, TStudent, TTeacher, TGraduateStudent) share the same TPerson object. This sharing of the TPerson object is within the TGradTeachingAsst object. No other class needs to know anything about the sharing.

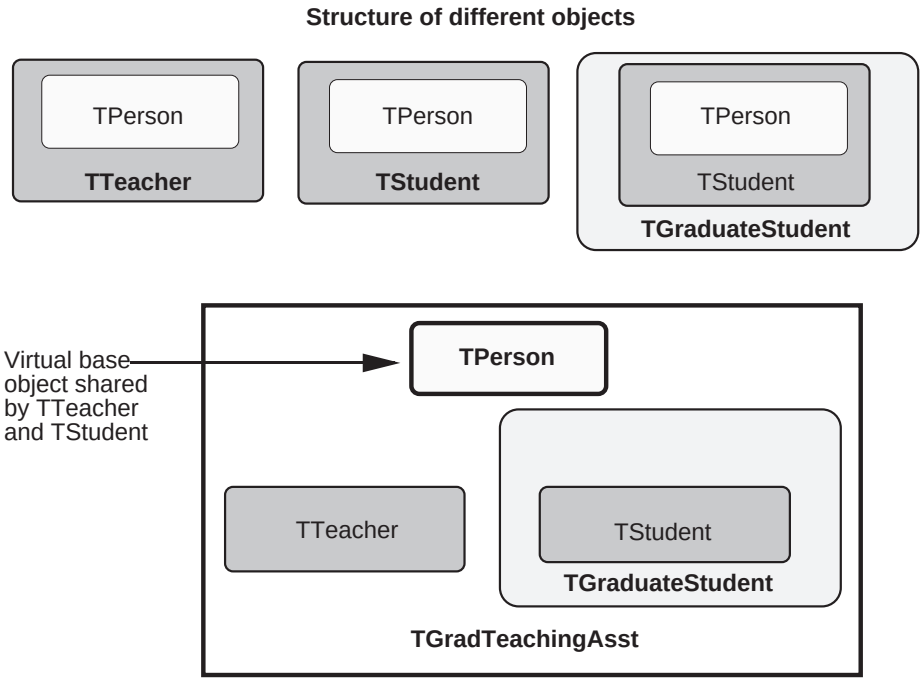


Fig. 6-11

Benefits of Virtual Base Classes

The advantage of this scheme is that clients of TGradTeachingAsst will always get the same behavior no matter how they access the TPerson object. Remember that TPerson, within a TGradTeachingAsst, can be reached from TGraduateStudent or from TTeacher. If the name or address within a graduate teaching assistant (TGradTeachingAsst object) is changed, then anyone using this TGradTeachingAsst object will see the new name (address). The old name (address) no longer exists in the object. This makes state management much simpler.

Virtual base classes also help in sharing information. Any data that should not be duplicated inside a complete object can be placed inside a virtual base class. The virtual base ensures that there is only one copy of its object in any complete object such as TGradTeachingAsst.

Virtual base classes help immensely in solving design problems where sharing information is essential. In many situations, the same class is inherited more than once indirectly. One cannot stop this from happening because such relationships are quite natural in inheritance hierarchies. Virtual base classes ensure sharing without the individual class designers having to worry about it (only the classes that directly inherit from the virtual base need to know about the sharing). This scenario is visible in C++ iostreams library implementations. Class ios (which holds the state of the stream) is a virtual base class for istream, ostream, and iostream, classes.

New Problems Due to Virtual Base Classes

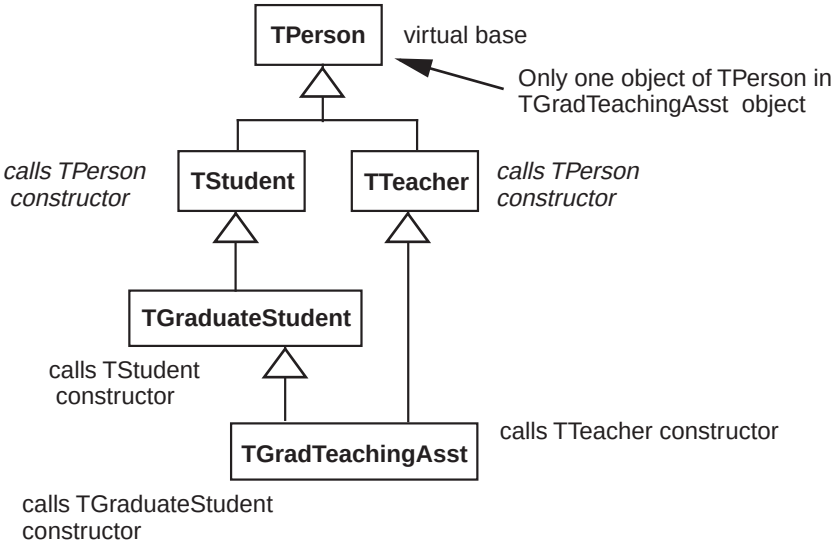
NOTE Much of the following discussion is very specific to C++. A reader not interested in language details may skip this section without any loss of continuity. But it is interesting to read this section from the viewpoint of language complexity and language design.

Virtual base classes are definitely useful when the designer uses it carefully. But a virtual base class also brings with it some new problems that never existed until now. The C++ language has solutions to these problems, but remembering the rules and implementing them correctly is not easy.

Problem of multiple constructor calls: In the original TGradTeachingAsst hierarchy, each derived class invokes the constructor of its direct base classes. So TStudent invokes TPerson constructor—the direct base of TStudent. Similarly, TTeacher also invokes TPerson constructor. On the other hand, TGraduateStudent invokes TStudent constructor (which in turn invokes the TPerson constructor). Finally, TGradTeachingAsst invokes TGraduateStudent constructor (which in turn invokes TStudent constructor) and TTeacher constructor (which in turn invokes TPerson constructor) (Fig. 6-12 and 6-13).

When we traverse the constructor call paths, it is easy to notice two calls to the TPerson constructor. And that is really confusing because in this hierarchy with virtual base class TPerson, there is only one TPerson object in any complete TGradTeachingAsst object. So how can the constructor execute *twice* for one object of TPerson?

Remember that we have not changed any implementation code in the TGradTeachingAsst hierarchy. We just added the virtual keyword to the base class declaration of TStudent and TTeacher. In the original hierarchy, without virtual base class TPerson, we really did create two TPerson objects (hence two calls to TPerson con-

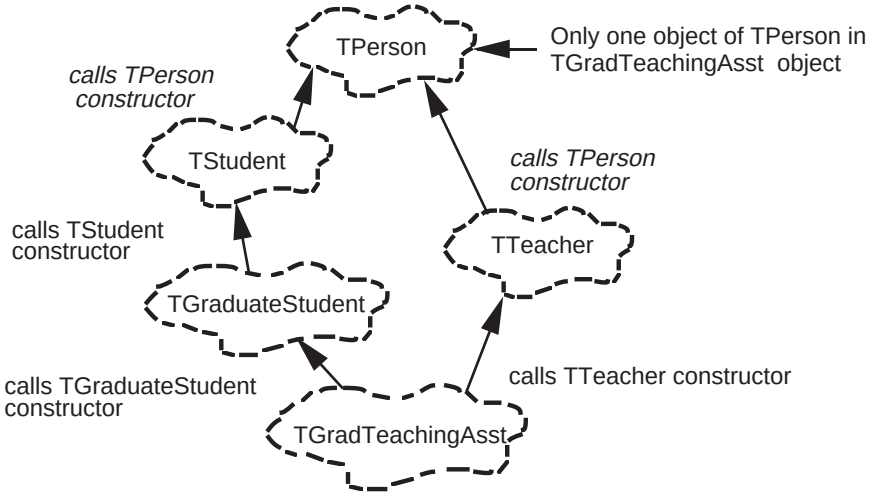


Repeated Constructor Call Problem with Virtual Base Classes

Fig. 6-12

structor). But that is no longer true in the new hierarchy with virtual base classes. The language must have some way of avoiding this duplicated constructor call.

As defined in the C++ language standard, it is the responsibility of the *most derived class* constructor to invoke the constructor for the virtual base class. In the above hierarchy, Fig. 6-12 or 6-13, TGradTeachingAsst is the final object or the *complete object*. In other words, TGradTeachingAsst is not created as part of another derived class—it is a



Repeated Constructor Call Problem with Virtual Base Classes

Fig. 6-13

stand-alone object. The TGraduateStudent object is a sub-object created as part of the TGradTeachingAsst object. In fact, four sub-objects (TGraduateStudent, TTeacher, TStudent, TPerson) are created within a TGradTeachingAsst object. The class of the completed object is the most derived class which is TGradTeachingAsst in this example. So every TGradTeachingAsst constructor must invoke an appropriate constructor for TPerson. In our implementation, we have a single constructor in the TGradTeaching-Asst class that must invoke the constructor of the virtual base class TPerson as part of its initialization phase.

A careful reader would have already noticed a major problem with this scheme. It is easy to understand the rule about the most derived class being responsible for invoking the virtual base class constructor. But the other classes in the hierarchy were written before TGradTeachingAsst class and those classes invoke the constructor of TPerson as part of their initialization. For example, TTeacher class invokes the constructor of TPerson and so does TStudent. The code to do this is already written. We cannot go back and change that code just because the TGradTeachingAsst class needs to call the TPerson constructor. One should not have to change an existing class when a new derived class is added—that is contrary to the principles of software reusability and encapsulation. Here is a code fragment from the TTeacher class that clearly depicts the situation.

```
TTeacher::TTeacher(const char theName[],
                   unsigned long theSSN,
                   const char theBirthDate[],
                   const char theAddress[],
                   Rank theRank,
                   EDepartment theDepartment)
              : TPerson(theName, theSSN, theBirthDate, theAddress)
{
  // and the rest of the implementation code
}
```

Does the virtual base class create more problems than it solves? The next few paragraphs will answer that question.

All virtual base classes are initialized by the constructor of the most derived class. When a complete object is being created, existing calls to the virtual base class constructor from the sub-object constructors are ignored.

In our example, when an object of TGradTeachingAsst is created, it is the complete object, and TStudent, TTeacher, and TGraduateStudent are all sub-objects (Fig. 6–14). The most derived class, TGradTeachingAsst should invoke the constructor of TPerson. In addition to that, TGradTeachingAsst also invokes the constructors for TTeacher and TGraduateStudent. With normal object creation, TTeacher's constructor would call TPerson's constructor and TStudent's constructor would also invoke TPerson's constructor. But when the TGradTeachingAsst constructor executes (remember that it is the most derived class), existing calls to the TPerson constructor from TStudent and TTeacher are ignored and only the call from TGradTeachingAsst constructor to TPerson constructor is used. In effect, when a TGradTeachingAsst object is created,

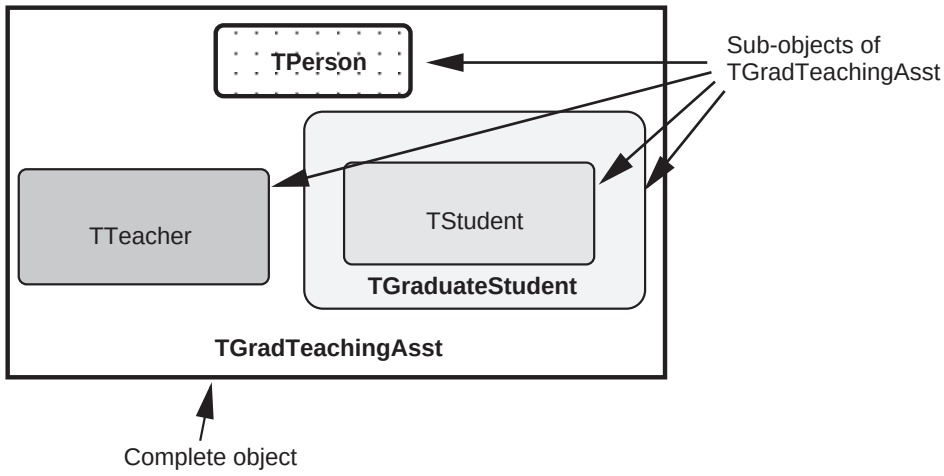


Fig. 6-14

the compiler knows that it is the complete object and constructor calls to the virtual base class object from other sub-objects are ignored. At least, the programmer need not worry about the issue. Note that in all these situations we never have to know the ordering of constructor calls. Programmers just need to make all the constructor calls and leave the responsibility of ordering the calls to the compiler. The compiler knows the correct order in which to invoke the constructors.

NOTE If the most derived class constructor fails to invoke a constructor of the virtual base class(es) explicitly, then the compiler will try to invoke the default constructor of the virtual base class(es), if there is an accessible default constructor in the virtual base class(es). Otherwise, a compile-time error is generated.

Anyone using virtual base classes must remember the above rule about virtual base class constructors. It would be easier to write code if the virtual base class(es) has a default constructor (one which can be called without any arguments) because then the compiler will make the right constructor calls; we don't have to remember all the details. This does not imply that every virtual base class must provide a default constructor. The decision to provide a default constructor (for that matter any other constructor) in a class is an interface design issue that cannot be compromised. A default constructor is logical only if the object of the class can be properly initialized, even without any arguments to the constructor. Programming with virtual base classes is a little easier when they (virtual base classes) have default constructors. But if adding a default constructor to a class compromises the interface of the class, then one should not provide it. In our TPerson example, it does not make any sense to provide a default constructor because there are no default values for a person's name, age, etc.—they must all be specified when a TPerson object is created.

☞ **Don't provide some member function just to make compiling code easier. It is more important to have abstraction integrity.**

After changing the TPerson to a virtual base, if a stand-alone TGraduateStudent object is created, then TGraduateStudent becomes the most derived class and it should call the TPerson constructor. In other words, the most derived class is the object of the class that we create explicitly. If we create a TTeacher object, then TTeacher is the most derived class and its constructor should call TPerson's constructor, which is already being done in TTeacher. If all these rules make you dizzy, don't worry, the compiler will flag the missing constructor calls and makes you fix the error. There is no way the error can get past the compilation stage and cause trouble at run time (unless of course the compiler itself has bugs in it).

With all this taken into account, here is the correct constructor of TGradTeachingAsst:

```
TGradTeachingAsst::TGradTeachingAsst(const char theName[],
        unsigned long theSSN, // of the graduate student
        const char theBirthDate[],
        const char theAddress[],
        EStudentStatus theStatus,
        EDepartment studentDepartment,
        const Teacher& advisorInCharge,
        EDepartment teachingDept)
    // Initialize direct base class TGraduateStudent
: TGraduateStudent(theName, theSSN, theBirthDate, theAddress,
              theStatus, studentDepartment, advisorInCharge),
    // Initialize direct base class TTeacher
  TTeacher(theName, theSSN, theBirthDate, theAddress,
         eGraduateTeachingAssistant, teachingDept),
    // Initialize the virtual base class TPerson
TPerson(theName, theSSN, theBirthDate, theAddress)
{
    // Implementation code for TGradTeachingAsst
}
```

By now, one must be thoroughly convinced about the complexity and rules of MI, and more so with virtual base classes. It is clearly evident that MI is not for the faint hearted. Using MI correctly and effectively requires very good knowledge of the language and design principles.

EIFFEL Under Eiffel, the solution to avoid duplication of features (method or instance variable) is a little different. It is not an error for a class to inherit repeatedly (even directly) from a single class. A feature (not an object) from the common parent class is considered shared in the final object if the feature has not been renamed along any of the inheritance paths. (Renaming was discussed earlier in this chapter.) This rule is quite powerful and easy to understand. When a class inherits from another class, it has to rename any feature that encounters a name clash. If a feature has not been renamed in the hierarchy, it guarantees that the feature has no name conflict anywhere in the hierarchy (otherwise it would have been renamed.) Any such feature would be shared in the object of the final class—it is not duplicated. In the example above, none of the features of TPerson (name, address, ssn, birthDate) would be renamed (there is no need to) in TStudent, TTeacher, and TGraduateStudent and, hence, would not be duplicated in TGradTeachingAsst. Hence, in Eiffel, TGradTeachingAsst need not do anything extra to retain only a single

copy of name, ssn, address, and birthDate. At first glance, this rule in Eiffel may not appear to be very different from the C++ virtual base classes in this particular example. But it is a little different and more powerful. For example, let's assume that a graduate teaching assistant would like to have two different mailing addresses—one for student related correspondence and another for graduate teaching assistant related correspondence. With this requirement, the declaration of TGradTeachingAsst in Eiffel would be:

```
     – Eiffel Code
class TGradTeachingAsst
     inherit
         TTeacher
         rename
             address as teaching_address

         TGraduateStudent
         rename
             address as student_address
     – Other details of class not shown
```

With this declaration, objects of TGradTeachingAsst will automatically get two copies of the feature address but one of them is referred as teaching_address and the other one as student_address. Other duplicated features (name, ssn, etc.) inherited by TGradTeachingAsst from TTeacher and TGraduateStudent are automatically shared because they have not been renamed anywhere along the paths leading to TGradTeachingAsst class. The advantage is that neither TTeacher not TGraduateStudent need to know anything about this duplicated feature—address. This is not done so easily under C++ without changing the classes TTeacher and TStudent.

Because of this renaming of features, Eiffel can easily handle directed repeated inheritance. For example , assume that class A has a feature X and class C inherits from A as follows:

```
class C
        inherit
             A – first inheritance from A
             rename
                 X as X_1

             A – second inheritance from A
             rename
                 X as X_2
```

Because of renaming, when C refers to the feature X in A, it will use the name X_1 or X_2. Hence, there is no ambiguity as to which A within C is being accessed.

Comparing MI in Eiffel and C++

The solution for duplicated objects is somewhat more elegant under Eiffel. Here, needless duplication of individual features (data members or methods) are eliminated automatically, whereas in C++, duplication of objects is eliminated by the programmer by using

virtual base classes. With C++, if we really want to retain multiple copies of some data members of a class (but not all of them), then there is no easy solution except to redesign the hierarchy such that only those that never have to be duplicated reside in the virtual base class. Duplication of features under Eiffel can be controlled at the feature level, whereas with C++, granularity of duplication control is limited to a class. In the above example, with Eiffel one can control the sharing (or replication) of each feature (data member) in TPerson, but in C++, we only have the option to share or replicate TPerson within TGradTeachingAsst objects.

Here is the corrected TGradTeachingAsst class, which initializes the virtual base class TPerson:

```
class TGradTeachingAsst : public TTeacher, public TGraduateStudent {
     public:
          TGradTeachingAsst(const char theName[],
                  unsigned long theSSN, // of the graduate student
                  const char theBirthDate[],
                  const char theAddress[],
                  EStudentStatus theStatus,
                  EDepartment studentDepartment,
                  const TTeacher& advisorInCharge,
                  EDepartment teachingDept);

          TGradTeachingAsst(const TGradTeachingAsst& copy);
          TGradTeachingAsst& operator=(const TGradTeachingAsst& assign);
          ~TGradTeachingAsst();

          // This method must be overridden
          virtual void Print() const;

          void SetStudentsDepartment(EDepartment dept);
          EDepartment GetStudentsDepartment() const;

          // This method must be overridden because
          // for a GTA the rank cannot be changed
          bool SetRank(ERank newRank);

          void SetTeachingDepartment(EDepartment dept);
          EDepartment GetTeachingDepartment() const;

};
```

And for completeness, here is part of the implementation:

```
#include <iostream.h>

TGradTeachingAsst::TGradTeachingAsst(const char theName[],
            unsigned long theSSN, // of the graduate student
            const char theBirthDate[],
            const char theAddress[],
            EStudentStatus theStatus,
            EDepartment studentDepartment,
```

```
            const TTeacher& advisorInCharge,
            EDepartment teachingDept)
        // Initialize direct base class TGraduateStudent
    : TGraduateStudent(theName, theSSN, theBirthDate, theAddress,
                theStatus, studentDepartment, advisorInCharge),
        // Initialize direct base class TTeacher
      TTeacher(theName, theSSN, theBirthDate, theAddress,
             GraduateTeachingAssistant, teachingDept),
        // Initialize the virtual base class TPerson
      TPerson(theName, theSSN, theBirthDate, theAddress)
     {
         // Implementation code for TGradTeachingAsst
     }

void
TGradTeachingAsst::Print() const
{
    // We get the appropriate pieces of data and print them
    EStudentStatus studentStats = GetStatus();
    EDepartment studentDepartment = TGraduateStudent::GetDepartment();
    EDepartment teachingDepartment = TTeacher::GetDepartment();

    cout << "Name: " << TPerson::GetName() << endl;
    cout << "This person is a Graduate Student" << endl;
    cout << " in the department of: " <<
         departments[(int) studentDepartment] << endl;
    cout << "The graduate teaching assistantship is in the department: "
         << departments[(int) teachingDepartment] << endl;
}

void
TGradTeachingAsst::SetStudentsDepartment(EDepartment dept)
{ TStudent::SetDepartment(dept); }

EDepartment
TGradTeachingAsst::GetStudentsDepartment() const
{   return TStudent:GetDepartment(); }

void
TGradTeachingAsst::SetTeachingDepartment(EDepartment dept)
{   TTeacher::SetDepartment(dept); }

EDepartment
TGradTeachingAsst::GetTeachingDepartment() const
{   return TTeacher::GetDepartment(); }

bool
TGradTeachingAsst::SetRank(ERank newRank)
{
    // Do nothing because the rank of a TGradTeachingAsst cannot be changed
    // It is always eGraduateTeachingAssistant which is already set in
    // the constructor call to TTeacher
    cout << "Error: Cannot change the rank of TGradTeachingAsst" << endl;

    return false;
}
```

GENERAL PROBLEMS WITH INHERITANCE

We have discussed inheritance in all its glory and usefulness in this chapter and also in Chapter 5. If you haven't already noticed, inheritance is a static relationship, meaning it is a relationship that does not lend itself to changes easily—it lacks flexibility. Furthermore, changing the relationships in a MI hierarchy is even more difficult. Inheritance is a great tool for modeling relationships found during domain analysis. But inheritance is not the best tool for situations where changes in relationships among classes need to be considered. The relationships in an inheritance hierarchy are decided and coded based on static relationships among classes. But making changes/additions to these relationships later is a difficult task. And this difficulty is felt even more with MI hierarchies. It is impossible to model dynamically changing relationships using MI. For example, TGradTeachingAsst is both a TGraduateStudent and a TTeacher at all times. This is a very rigid relationship. In reality, a person plays the role of a TTeacher when she is teaching some course and she behaves as a TGraduateStudent while attending graduate school. Strictly speaking, she exhibits the characteristics of a TTeacher at times and TGraduateStudent some other time but she is not both at the same time. This property of a TGradTeachingAsst is not reflected in the hierarchy that we have seen so far.

Taking a different approach, enrolling for courses is a capability that a TPerson can acquire. Similarly, teaching courses is also a capability a TPerson can acquire after going through some certification process. And a TGradTeachingAsst acquires both capabilities. This looks correct until we consider adding a new capability to any TPerson. For example, consider what happens if we want to add a research assistant capability to a TPerson. A research assistant need not be a student. One could extend the hierarchy as follows in Fig. 6-15:

New class TResearchAsst added to the existing hierarchy

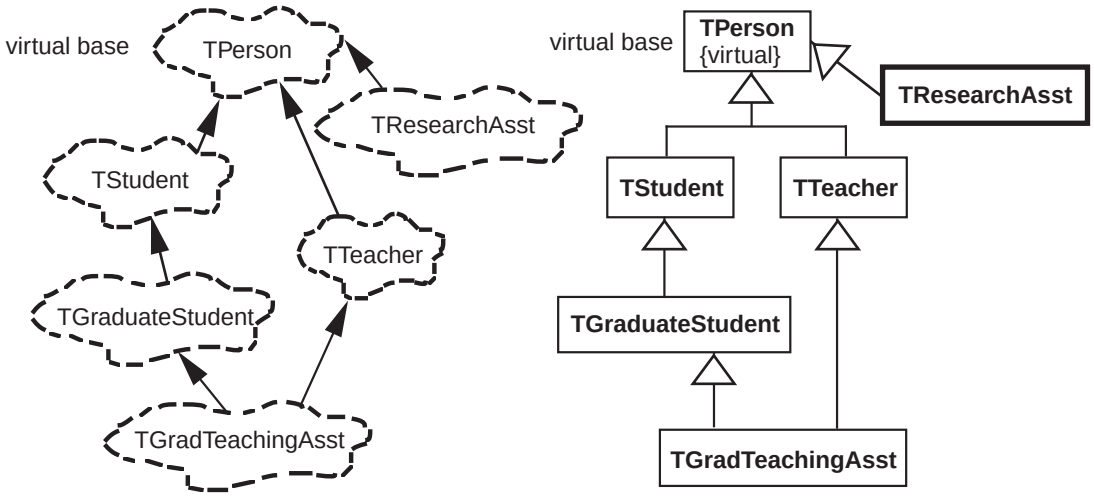


Fig. 6-15

That looks like a reasonable design—any TResearchAsst is also a TPerson. What if a TGraduateStudent also takes on the job of a TResearchAsst, probably in a different (or even in the same) department. With all our experience with inheritance, can we say that TGraduateStudent is also a TResearchAsst? Definitely not because *not all* TGraduateStudents are TResearchAssts. The problem is that performing research duties is a capability that some TPerson can acquire without being a student or a teacher. A student (or even a teacher) can also be a research assistant but that is not always true. Inheritance is suitable for modeling relationships among classes that are always true. Definitely a TGraduateStudent is always a TStudent - nothing wrong with that. But how can we model the situation of a TGraduateStudent who might also be a TResearchAsst? And can we add another class TEmployee to this hierarchy to model employees in the university? Teachers and research assistants are employees, and a student could also be employed in the university. But not all students are employees. We encounter conflicting requirements, and inheritance relationships are no longer suitable to model such complex relationships. We need to look elsewhere for more meaningful solutions.

The scenario depicted in the previous paragraph is a common situation in commercial software development. We encounter the necessity to add different capabilities to different classes. There can be many such capabilities and it should be possible to pick one or more of them. We discuss two possible solutions for adding capabilities to classes. One of them is a more static relation and the other one is more suited for dynamically changing environments.

USING MIXIN CLASSES FOR ADDING STATIC CAPABILITIES

In our university example, we can easily notice the fact that teaching a course, qualifying to register for a course, performing research duties, etc. are capabilities a TPerson can acquire. Moreover, a person can acquire any combination of these capabilities. For example, a person could be a student in the Electrical engineering department with research work in Aerospace Engineering. Similarly, a person could be a student in the Computer Science Department with a graduate assistantship in the Mathematics Department and research duties in the Physics Department. These situations show different combinations of capabilities (or roles). A mixin class is very handy to model such scenarios.

Definition of a Mixin Class

A mixin class adds new capabilities to other classes. One never creates an instance of a mixin class. It does not make any sense to create an instance of the mixin class because the mixin class adds *flavor* to some other class—it is to be mixed in with another class. In this book we shall use a unique notation to represent mixin classes—they always start with the letter 'M'.[7] By using mixin classes, one can combine different capabilities (flavors) to form new capabilities. This is very similar to adding toppings to ice cream. At an ice cream parlor, the customer can choose any combination of toppings. I might pick

[7]As you already know, normal classes begin with 'T'.

cherries, nuts, and cookies. Someone else might opt for berries and chocolate syrup. One has the freedom to make ice creams with any combination of toppings. There is no restriction on the combination of toppings. Each topping can be viewed as a mixin class that adds a new flavor. Mixin classes afford this freedom to application designers. We are going to make use of mixin classes to solve our university problem.

NOTE Mixin classes represent static relationships. They allow us to add capabilities when designing hierarchies. We cannot add capabilities to an object dynamically (at run time) using mixin classes. A scheme for adding capabilities dynamically is discussed later in this chapter.

The ability to become a student in the university system can be captured in the mixin class MCanBecomeStudent. This mixin adds methods to register for courses, set student identification, etc. Most of these methods would be pure virtual, making the mixin an abstract base class.

```
enum EQualification { eNoSchooling, eHighSchool, eUnderGraduate, eGraduate,
                                    eDoctorate };

class MCanBecomeStudent {
     public:
          // Constructors and other usual functions not shown
          void SetDepartment(EDepartment dept);
          EDepartment GetDepartment() const;

          virtual bool EnrollForCourse(const TCourse& aCourse) = 0;
          virtual bool DropFromCourse(const TCourse& course) = 0;
          virtual void ListCoursesRegisteredFor() const = 0;
          virtual EQualification GetStudentQualification() const;
          // and many more
};
```

Given this new class, we can change our student-teacher hierarchy as follows (see Fig. 6-16).

New hierarchy using mixin classes

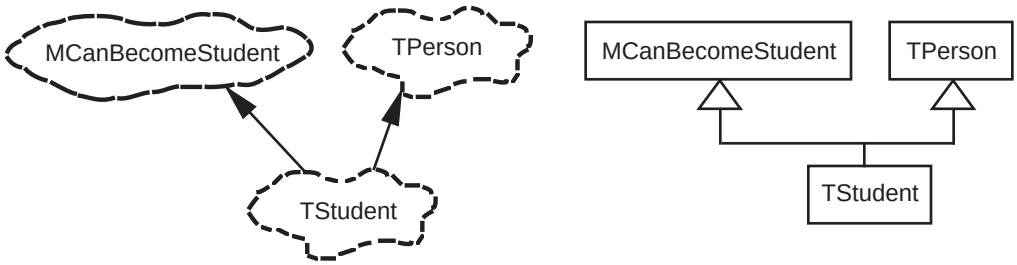


Fig. 6-16

Now, a student is any person who acquires the capabilities to become a student. Class TStudent must implement all the pure virtual methods inherited from MCanBecomeStudent. It is also possible to provide all the (default) implementation in MCanBecomeStudent and make it optional for TStudent to override them. The decision to make a method pure virtual in a mixin class is entirely dependent on the services provided by the mixin class.

The next step is to add a mixin class that captures the capability to teach a course in the university. A person can teach a course if she is qualified to do so. The ability to teach is captured in a new mixin class MQualifiedToTeach. This class will provide methods to manipulate courses taught, supervisor, department, etc.

```
class MQualifiedToTeach {
     public:
          // Usual constructors, etc. not shown for simplicity
          virtual void SetDepartment(EDepartment dept);
          virtual EDepartment GetDepartment() const;
          virtual void ListCoursesQualifiedToTeach() const = 0;
          virtual double GetYearsOfExperience() const;

          // What degree (master, Phd, etc) does the Teacher have?
          virtual EQualification GetHighestDegree() const = 0;
          virtual TPerson GetSupervisor() const = 0;
          virtual double GetSalary() const;
          // and many more
};
```

Method ListCoursesQualifiedToTeach shows all the courses the person has been certified to teach. This list of courses might be setup during object creation and might also be updated later through other methods. It is also possible that the university keeps a database of people with their certification history. Salary is the compensation paid to the person for teaching the course(s).

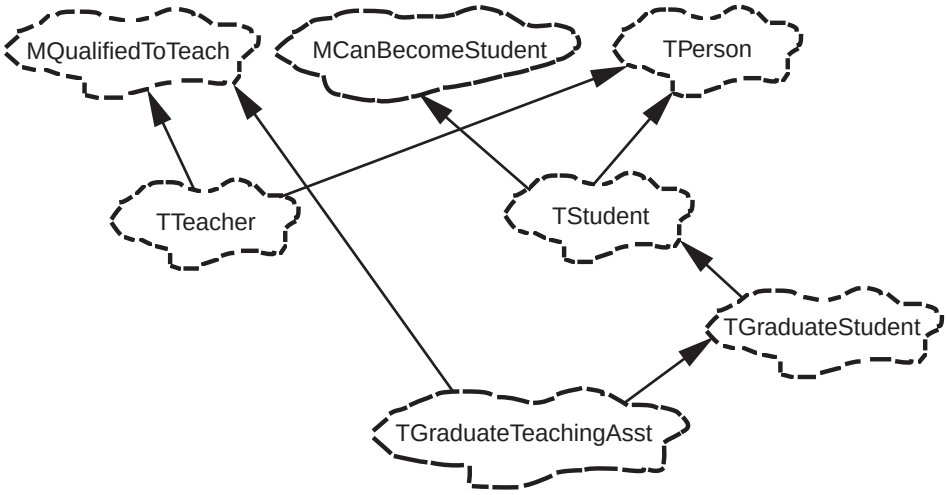
The new hierarchy is shown in Fig. 6-17.

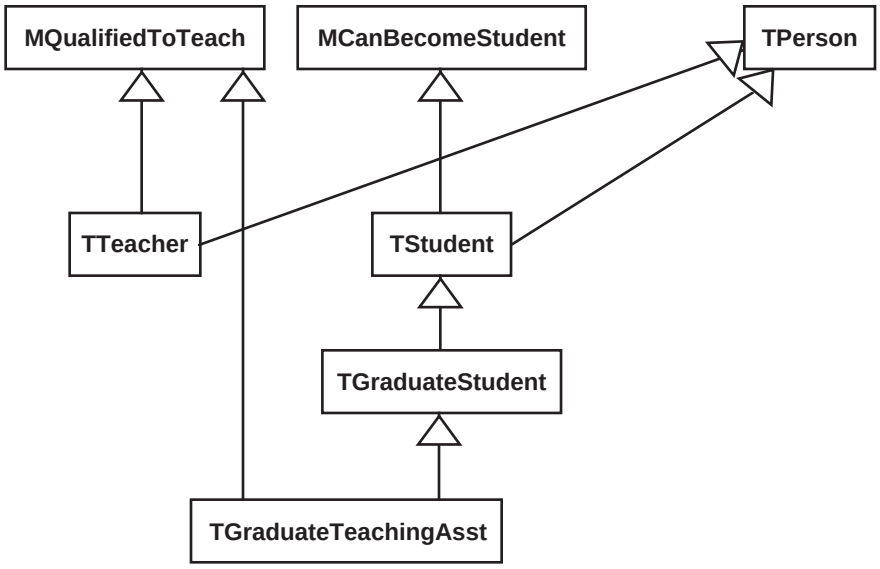
Along the same lines, we can add another mixin class to account for people who are involved in research. Let's call this mixin class MQualifiedToDoResearch. This class captures the essential features of a person doing research.

```
class MQualifiedToDoResearch {
     public:
          // number of years of research experience
          virtual double GetResearchExperience() const;
          // number of papers published
          virtual int GetNumberOfPublications() const;
          virtual EDepartment GetDepartment() const;
          virtual EQualification GetHighestDegree() const;
          // and other methods
};
```

With this new class, it is easy to model a TResearchAsst. This class inherits from the mixin MQualifiedToDoResearch and TPerson. A graduate research assistant will be another class inheriting from TGraduateStudent and TResearchAsst (see Fig. 6-18 and 6-19).

New hierarchy using mixin classes





New hierarchy using mixin classes

Fig. 6-17

New hierarchy using mixin classes

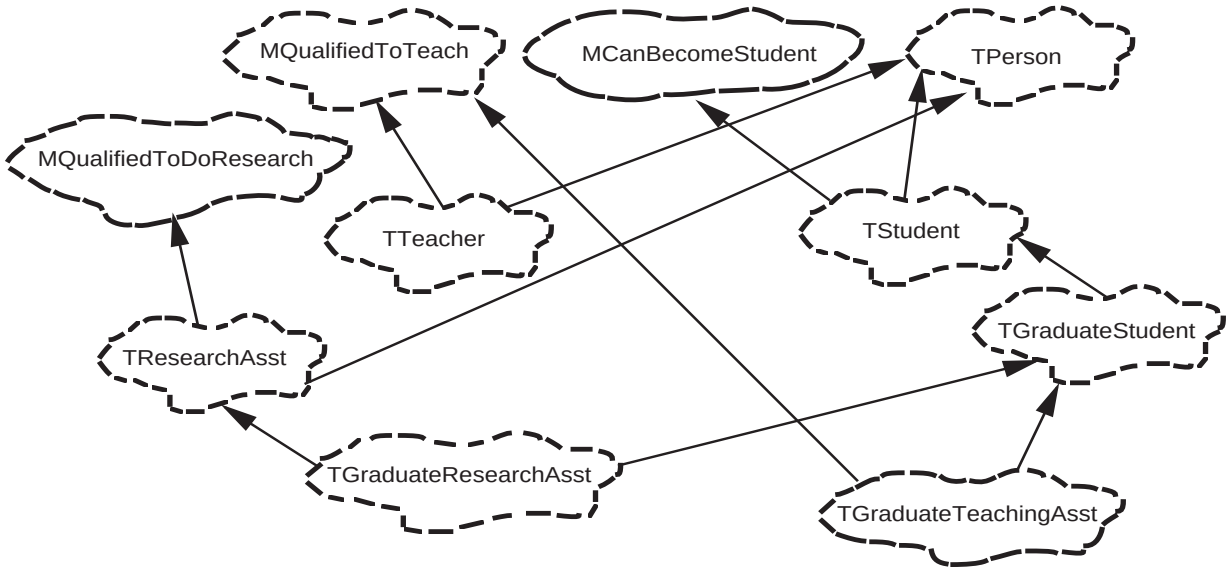


Fig. 6-18

NOTE In this example of mixin classes (Fig. 6-19), the reader might have noticed that no implementation code is shown. It is more important to understand the concept of mixin classes. The implementation code is not very hard to write once the design concepts are clear. Hence, the emphasis is on concepts rather than code.

This new hierarchy exemplifies the power of mixin classes. They allow the designer to provide orthogonal capabilities that can be mixed together in any combination desired. One does not have to pick a compromise solution (which is often the case with simple MI). This design gets rid of the virtual base class TPerson. We ran into a number of complex design issues and code management issues because of the virtual base class TPerson. Mixin classes help in eliminating virtual base classes. The original MI hierarchy with the TGradTeachingAsst looks like a diamond. These diamonds are a potential source of many problems when designing for flexibility. Mixin classes provide an alternative in such situations. The flexibility and simplicity of design is achieved because the capabilities of a person are now decoupled. In the original design, a teacher was a person and the teaching capability was embedded in the class TTeacher. Another problem with the original design was that of rigidity. It is almost impossible to add new classes to the hierarchy to allow for new capabilities. Moreover, adding a new class will affect every class in the hierarchy. The mixin class hierarchy takes care of that problem. We can add new capabilities without affecting other classes in the hierarchy (as was done with MQualifiedToDoResearch).

New hierarchy using mixin classes

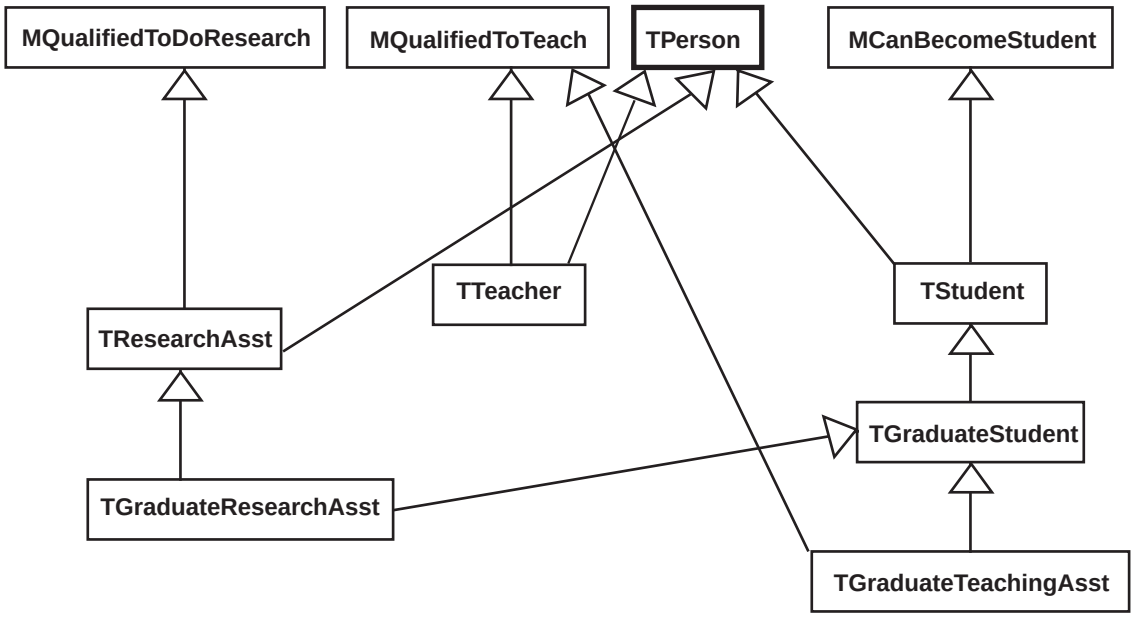


Fig. 6-19

When to Use Mixin Classes?

1. When there are a number of independent characteristics that any class can acquire, it makes sense to represent the characteristics as mixin classes. See MQualifiedToDoResearch and others above.

2. When a new capability needs to be added selectively to some classes in an existing hierarchy, mixin classes may be the only alternative. Existing classes that need the capability would inherit from the new mixin class but other classes that don't require this capability don't have to worry about the new mixin class.

NOTE Mixin classes need not always be used with public inheritance. As illustrated later in this chapter, a mixin class can be useful even under private derivation. For further discussion of this concept please refer to the section on private derivation later in this chapter.

DESIGNING FOR DYNAMICALLY CHANGING SITUATIONS

The mixin design presented above is definitely better than our original design using virtual base classes. But it is still not flexible enough. To understand the drawbacks of the new design, let's review a typical run-time scenario.

We want to minimize duplication of data at all times. That's why we used virtual base classes. Duplication of data not only causes wastage of resources but also causes data management headaches. So avoiding duplication of data is a noble goal. We also want to copy minimum data when objects are copied or when they need to change. For example, if a normal student in the university becomes a graduate student, we want to be able to modify (add) information that only affects the student related data. Definitely, there is no need to tinker with the data in TPerson. In other words, when a TStudent becomes a TGraduateStudent we should be able to just add the TGraduateStudent part of the data to TStudent. Along the same lines, when a TGraduateStudent becomes a TTeacher, we would like to throw away the TGraduateStudent part and add the TTeacher capabilities, without modifying the TPerson part. These seem like easy goals to achieve but one careful look at the object management scenario reveals the underlying problems.

When we have a TStudent object, how do we convert that object to a TGraduate-Student object? We have to create a new TGraduateStudent object using the data in the TStudent object. There is no simple way of doing this. We have to extract each and every field from the TStudent object and use that information (and the information provided by the client for the TGraduateStudent object) to create a new TGraduate Student object. Then we have to discard the TStudent object. The code fragment might look like this:

```
// This function creates a new TGraduateStudent object from an existing
// TStudent object. It deletes the original TStudent object after the new
// TGraduateStudent object is created.
TGraduateStudent*
CreateNewGraduateStudentFromStudent(TStudent* adoptStudent,
                    EDepartment newDepartment,
                    TTeacher& theAdvisor)
{
    // We extract the necessary information from the existing TStudent
    // object (pointed by adoptStudent), and pass that information to
    // TGraduateStudent constructor.
    TGraduateStudent *gp
        = new TGraduateStudent(adoptStudent->GetName() // get name part
            adoptStudent->GetIdentificationNumber(), // Student ID
            adoptStudent->GetBirthDate(), // birth date from TPerson
            adoptStudent->GetAddress(), // address from TPerson
            adoptStudent->GetStudentStatus(), // status from TStudent
            newDepartment,           // department for the grad student
            theAdvisor);

    // Next delete the TStudent object
    delete adoptStudent;

    return gp;
}
```

It is interesting to note that even though the TPerson part of the TStudent remains the same in the TGraduateStudent, we still couldn't reuse the data directly. We had to create a new TGraduateStudent (which created a new TStudent and a new TPerson) and

then throw away all the existing useful pieces of information in the original TStudent object.

NOTE There is another solution possible for this scenario. Class TGraduateStudent could provide a constructor that accepts a TPerson object. This constructor would create a new TGraduateStudent object using the information provided in the TPerson argument. This principle can also be applied to TTeacher class. This solution doesn't prevent the duplication of data because the new object created would copy the information from the TPerson object. The user must delete the TPerson object if it is no longer needed. Needless to point out that a TStudent can be substituted in place of the TPerson argument to the constructor. Generalizing, any derived class of TPerson can be passed to this special constructor. This scheme is used quite often in the industry. In general, a derived class must provide constructors to accept base class objects, if there is a requirement to create derived class objects from existing base class objects. This can be further extended to assignment operators also (derived class assignment operator accepting a base class object as the argument).

Another approach could be to create an empty TGraduateStudent object and then copy in the information from TStudent, as follows in Fig. 6-20.

```
// This function creates a new TGraduateStudent object from an existing
// TStudent object. It deletes the original
// TStudent object after the new TGraduateStudent object is created.
TGraduateStudent*
CreateNewGraduateStudentFromStudent(TStudent* adoptStudent,
                  EDepartment newDepartment,
                  TTeacher& theAdvisor)
{
    // Create an empty TGraduateStudent object
    TGraduateStudent *gp
        = new TGraduateStudent(0 // no name specified
              0, // Student ID not specified
              0, // birth date not specified
              0, // address not specified
```

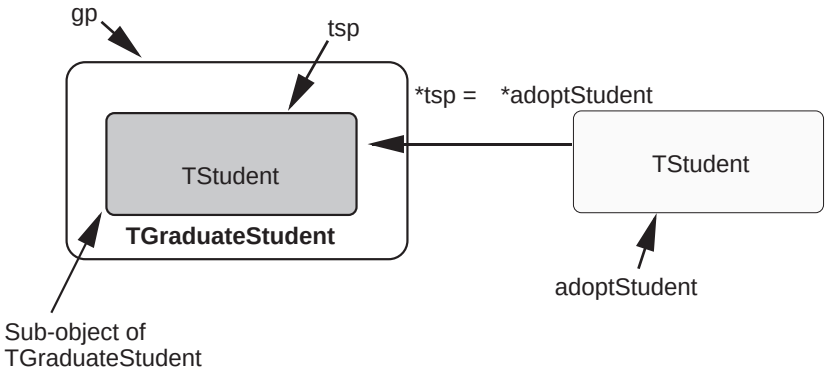


Fig. 6-20

```
            eUnknown,              // status not specified
            newDepartment,         // department for the grad student
            theAdvisor);

    // Valid Initialization because TGraduateStudent is a TStudent
    TStudent* tsp = gp;

    // Next copy all the essential information from TStudent. Dereferencing
    // tsp (which is a TStudent pointer) gives access to a TStudent object.
    // But since tsp actually points to a TGraduateStudent object, we are
    // assigning to the TStudent part of a TGraduateStudent
    *tsp = *adoptStudent;
    // Assignment from old TStudent to new TGraduateStudent. After this
    // assignment, the department part of the TGraduateStudent needs to
    // reset.
    gp->SetDepartment(newDepartment);
    // The advisor part is not accessed by TStudent, so it is fine.
    // Next delete the TStudent object
    delete adoptStudent;

    return gp;
}
```

This scheme still doesn't avoid the copying of data from old TStudent to the new TGraduateStudent object. It would have been perfect if we could somehow add the extra information about TGraduateStudent (like advisor) to the existing TStudent and convert it to a TGraduateStudent. We can only hope for that but it is not possible with any of our MI hierarchies discussed earlier (no matter whether it is C++ or Eiffel).

CAUTION If the existing TStudent object should not be deleted by the function CreateNewGraduateStudentFromStudent, then do not use this function. If the intention is to create a TGraduateStudent from a TStudent but without destroying the TStudent object, one could write an overloaded function CreateNewGraduateStudentFromStudent that accepts a reference to TStudent or we can add a bool argument (with a default value) to the existing function that indicates the client's intention. Whatever implementation is chosen, ensure that the client understands the consequences with clear documentation and argument types.

This scenario shows the inflexibility of MI for dynamically changing scenarios. What we just discussed is a very common situation in databases.

When a TGraduateStudent becomes a TTeacher, the scenario is even worse. The only common piece of information here is the TPerson part. But we still have to copy it into a new TTeacher object as follows.

```
// This function creates a new TTeacher object from an existing
// TGraduateStudent object. It deletes the original TGraduateStudent object
// after the new TTeacher object is created. Follows the same scheme as the
// one shown above for creating a TGraduateStudent from TStudent

TTeacher*
CreateNewTeacherFromGradStudent(TGraduateStudent* adoptStudent, EDepartment
                                        newDepartment, ERank theRank)
```

```
{
    TTeacher *tp = new TTeacher( 0, // name
              0,                // Id
              0,                // Birth date
              0,                // address
              theRank,
              newDepartment     // department for the grad student
              );

    // Next copy all the essential information from TGraduateStudent
    TPerson* p = tp; // again valid because a TTeacher is a TPerson
    *p = *adoptStudent; // This will copy the TPerson information

    delete adoptStudent;
    return tp;
}
```

Here again, there is quite a bit of data copying and deletion of useful information. This is a problem for which there might be a better solution.

Another problem is with multiple capabilities. Imagine what happens if a person is a graduate student in one department but a research assistant in another department. To manage these situations we have to create two independent objects—one for the graduate student and another for the research assistant in the other department. There is quite a bit of information (about the TPerson) duplicated here. And what if the same person is a student in two different departments (which is a very normal situation)? The scenario gets worse as we add more capabilities to the hierarchy.

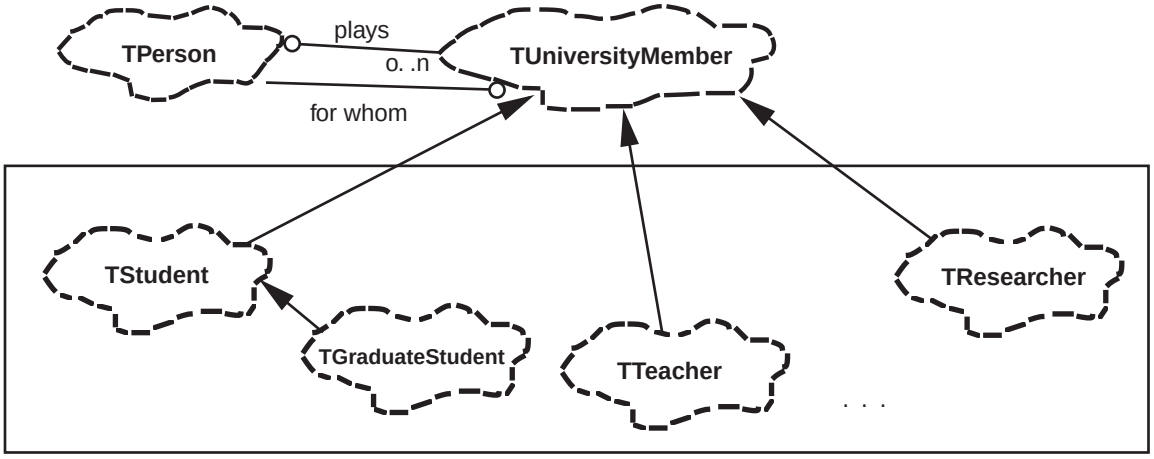
We are assuming that every person has only one *role* in the university system, which is completely wrong. A person is a student in only one department or a teacher in some other department but not both. In reality, we play many different roles. There is nothing unusual about a teacher of biochemistry wanting to become a student in computer science. And a student of finance could be an instructor in the athletics department. MI with virtual base classes is definitely not suitable for this situation. And the mixin hierarchy also fails because one needs to identify all possible combinations of leaf classes (such as TStudent, TStudent+TTeacher, TStudent+TResearcher, which will cause a combinatorial explosion of classes leading to a very complex (and yet very fragile) hierarchy.

We need to recognize the fact that a person plays many roles, but only one role is active at any time. Even though a person could be a teacher and student in the same university, she will be performing only one of the roles at any given time and not both. This is a normal trait of human beings. We play roles of father, manager, gardener, church choir singer, etc. in our daily life but not more than one at a time. The common feature of teacher, student, research assistant, graduate student, etc. is the fact that they are all roles played by a person. Once we recognize this key characteristic, it is easy to organize the hierarchy as follows in Fig. 6-21.

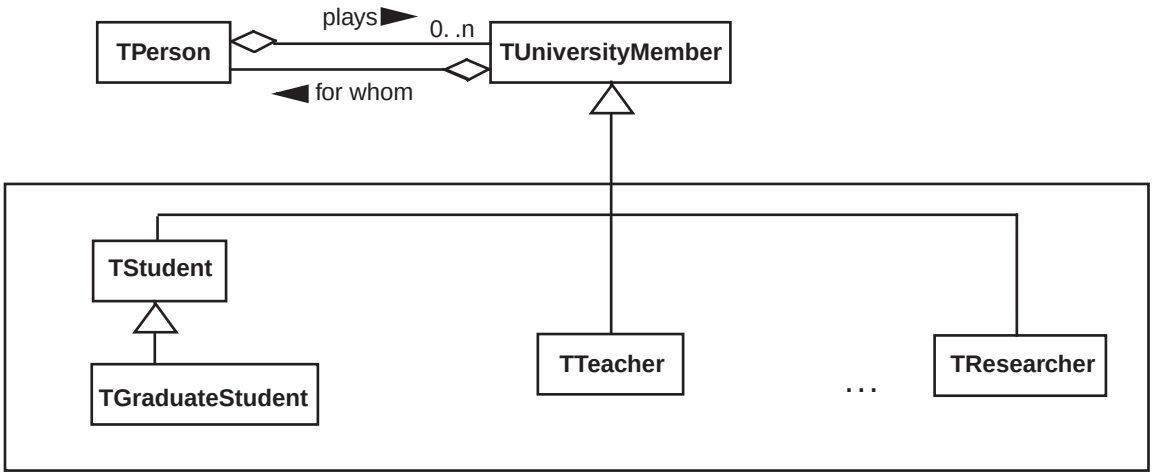
In this figure, every person has 'n' TUniversityMember roles. Each role belongs to only one person, as implied by the *for whom* relation. We can ask each TUniversityMember object as to whom this role belongs. Since a TPerson maintains a list of roles played, it is possible to enumerate all the roles played by a person.

What we have done is to decouple the roles from the person playing them. In that sense, it is similar to the mixin classes. However, it is very different from the mixin

A person plays many roles



Different roles played by a person



Different roles played by a person

Fig. 6-21

classes because the roles form a separate hierarchy. One can add any number of roles to a person, even the same role twice, such as student in more than one department. At any time, only one of the many roles played by the person is active. The TPerson class maintains a list of all roles played by the person. The individual roles maintain information relevant to that role only. So a TTeacher only keeps the information related to the teaching assignment (just as with mixin classes). Since all roles are attached to a person, there is no duplication of data about the person (such as name, address, etc.).

Design Flexibility of Role Playing Classes

Does this new hierarchy solve the problem of dynamically changing scenarios? Yes, it does it very well. For example if a person maintains three roles, say a teacher in one department, a student in another, and a researcher in yet another, we have a TPerson object with three TUniversityMember objects. If this person stops doing research, we need to reflect this change in the roles played. All we have to do is to discard the role TResearchAsst. Nothing else about the person has changed. Similarly, if a student role needs to be converted to a graduate student, we create a new TGraduateStudent object and assign the existing TStudent to the new TGraduateStudent (as we did earlier), but now the information to be assigned is highly diminished because there is no information about the TPerson in TUniversityMember. It also takes less time to create (and copy) any of the TUniversityMember subclass objects because the information that needs to be put into these objects is quite less. Compared to the previous designs, there is no extra memory consumed by this design because the information that is stored in any of the TUniversityMember base class objects is the same (all relevant information is in the derived classes of TUniversityMember). In fact, when one looks at the complete picture of managing changing roles, this new design requires somewhat less memory because it avoids data duplication completely.

How to Use Role Playing Classes?

It is time we understand how this new design works in reality. Class TPerson needs to support a new method AddRole. This method adds a new role to the existing set of roles (if any) for the person. We need another method SetActiveRole, which specifies the currently active role. For example, when dealing with a student who is also a researcher, we would set the active role to be the student role when manipulating the students attributes. Even though a person has many roles, only one of them is active at a given time.

The next problem is that of invoking specific methods on a TUniversityMember object. A TStudent, TTeacher, TResearchAsst, etc. have different methods but they have a common base class TUniversityMember. When we ask a TPerson object for the current active role, it will return a TUniversityMember object. We cannot apply methods from TTeacher on a TUniversityMember object because TUniversityMember is not a TTeacher (but a TTeacher is a TUniversityMember). So how do we overcome this problem?

This is a classic problem of type matching. Given a TUniversityMember object, how can we check if it is really an object of one of the derived classes? Furthermore, is it possible to convert (safely) a TUniversityMember object to the correct derived class object? All these problems have been encountered by object-oriented software developers time and again and fortunately there is a solution to this problem.

RUN-TIME TYPE IDENTIFICATION (RTTI)

RTTI is a comparatively new feature of C++. It allows programmers to safely (and in a portable manner) convert pointers and references to objects from one type to another. RTTI is clearly defined and is part of the language specification.

One of the most important features of RTTI is the **Dynamic Cast Operator.** This new operator allows the programmer to convert (if possible) a pointer of one type to a pointer of another type, at run-time. If the conversion fails, the result is 0. For example, if we have a pointer to TUniversityMember object and we would like to convert this pointer to a TTeacher pointer (if the TUniversityMember is really a TTeacher object), the following code would be appropriate:

```
void f(TUniversityMember* rp)
{

    // Is this pointer (rp) to TUniversityMember, really a pointer to
    // TTeacher?
    TTeacher* tp = dynamic_cast<TTeacher*> (rp);
    if (tp != 0) {
          // Yes, the object pointed by rp is a TTeacher object.
          // Perform any operations on the object using tp
    }
    else {
        // No. The object pointed by rp is NOT a TTeacher object
        // It is something else. Now tp is 0
    }
}
```

This conversion (or casting) of a pointer to a base class object to a pointer to a derived class object has been traditionally called *down casting*. Going from a base class to a derived class is seen as going *down* the class hierarchy and hence the name. Such down casts were done by using ad-hoc schemes (by keeping home grown type information in objects) and was very dangerous. RTTI makes it safe. Performing an unchecked cast from a base class to a derived class is similar to leaping off a cliff with the hope of finding a soft landing platform below; you are either successful or you end up breaking many bones! RTTI doesn't allow you to jump unless there is a well formed platform below. Here is the syntax of the dynamic_cast operator.

```
dynamic_cast<Typename> (pointer)
```

The name within < and > is the type name to which you want to convert the pointer to. If the pointer really points to an object of the requested type, then the conversion takes place. Otherwise, the dynamic cast operator returns a 0 pointer. The name inside the ( ) is the pointer undergoing the test and conversion. In the sample code above, we wanted to cast the pointer rp to the type TTeacher.

The dynamic cast operator executes some implementation code to verify the validity of the conversion. In other words, there is some run-time cost associated with

this operation—it is not a compile-time operation. It is also called a safe cast because the dynamic cast operator verifies the validity of the conversion and will return 0 if the attempted conversion is not possible. Hence, invalid implicit cast operations are not possible. Any cast operation will only succeed if the object is of the correct type. There is no guess work or blind leap of faith. Hence, it is safe to use.

Dynamic cast also works for object references, but the usage is a little different.

```
void f(TUniversityMember& refRole)

    // Is this reference to TUniversityMember really a reference to
    // TTeacher
          try {
          TTeacher& teacherRef = dynamic_cast<TTeacher&> (refRole);
          // Yes, the object referred by rp is a TTeacher object.
          // Perform any operations on the object using teacherRef
             }

    catch (bad_cast& b) {
          // No. The object referred by refRole is NOT a TTeacher object
          // It is something else. bad_cast is a pre-defined class used
          // by the language
    }
}
```

A pointer that is valid can never be zero. It is easy for the dynamic cast operator to return 0 when the cast fails. But for a reference to an object, there is no such distinguishing value to indicate an invalid reference. References are aliases to existing objects and a reference is always valid. Therefore the dynamic cast operator indicates failure by raising the bad_cast exception.[8] In anticipation of this situation we *catch* the exception. If the catch block is entered, then it is clear that the checked cast failed.

Dynamic casts work for *polymorphic* types only (i.e., for classes with virtual functions). The RTTI information is available for objects of classes with at least one virtual function. Dynamic cast will not work for objects that do not have any virtual functions. This is not really a problem because in most classes, if not any other method, the destructor is a virtual function making the class a polymorphic type. A polymorphic type is any type that can exhibit polymorphism. Dynamic casts work for virtual base classes and void pointers also. One can attempt to cast a void pointer to a pointer to some other class.

The typeid() operator

Dynamic cast internally uses a more powerful and general operator called typeid(). This operator returns type information about a specific type. It can also return the type information about an object whose type is not known exactly.

[8]Exception management is discussed in chapter 10. For now, it is enough to understand that it is an error condition that must be handled immediately or else the program will abort.

For example,

```
void f(TUniversityMember& roleRef)
{
    if (typeid(roleRef) == typeid(TTeacher) {
          // The object is really a TTeacher
    }
    else {/* It is not a reference to a TTeacher object */ }
}
```

typeid() returns a type_info object (described below) for the actual object. If two objects are the same type, then their typeids will compare equal. typeid() operator can be used for polymorphic types as well as non-polymorphic types. It works for language defined types as well as programmer defined types. Also note that typeid works for type names as well as object names.

```
// Built-in types
double di = 22.33;

double& dr = di;
if (typeid(dr) == typeid(double)) // NOT typeid(double&)
{     /* Do something */ }
```

typeid() operator is based on the class type_info.

```
class type_info {
    private:
        type_info(const type_info& );
        // cannot be copied by users
        type_info& operator=(const type_info&);
        // implementation dependent representation
    public:
        virtual ~type_info();
        bool operator==(const type_info&) const;
        bool operator!=(const type_info&) const;
        bool before(const type_info& rhs) const;[9]

        // Returns a pointer to the name of the type
        const char* name() const;
};
```

Objects of this class cannot be copied because the run-time system maintains these objects and uses them as needed.

CAUTION

RTTI is not a solution for all problems. It is useful (mostly) in situations where a down cast operation is needed. Such down cast operations are not very frequent. Most type dependent operations can be easily handled by using virtual functions. One should not write code that explicitly depends on the RTTI information. Not only is it expensive, but RTTI also makes your code less extensible.

[9]bool is language defined type in C++, which can be either "true" or "false."

The C++ RTTI mechanism also provides a static_cast mechanism for performing casts. static_cast<T>(e) converts the value e to type T, provided an implicit conversion exists from the type of e to type T. But, static_cast cannot cast away const. There is also a const_cast<T>(e) operation to convert a const or volatile. Here T differs from the type of e only in the presence or absence of const//volatile. The reinterpret_cast can be used for implementation-defined casts. It is identical to the old style (T)(e) casts. But reinterpret_cast cannot cast away const. Use these casts instead of the old style (T)(e) casts. These are safer, easy to locate in code if modifications are required, and they clearly convey the intention of the programmer. Refer to the C++ language reference manual for details.

The solution is based on the concept of run-time type determination. What we need is a way to query (and probably convert) the exact type of a given object. Given a TUniversityMember object, we should be able to determine its exact type (TTeacher, TStudent, etc.). All this is made possible by *Run-time Type Identification*.

EIFFEL Support for querying the *real* type of an object (and an object's internal structure) is supported in Eiffel by the predefined class INTERNAL. Any class that requires such facilities should inherit from the class INTERNAL.

SMALLTALK Facilities are available in Smalltalk to query the type of an object (class), checking if an object responds to a message (respondsTo), etc.

Now we are ready to continue with the university management problem and analysis of role playing classes. Once we get the current role, we can use RTTI to determine if it is of the required type (TTeacher, TStudent, etc.). If the TUniversityMember object is what we want, then we can use checked cast operator to down cast it (the TUniversityMember object) and safely perform all operations required. We can also provide methods in TUniversityMember to query if it is of a given type (one of TTeacher, TStudent, etc.). This can be done using type_info objects. With these modifications, the TPerson class would look like this:

```
/* The new TPerson class. All details are not shown. Only the new additions
   related to the role playing concept are shown. All TUniversityMember
   objects are owned by TPerson. Whenever a new role is created and added
   to a TPerson object (using the AdoptNewRole method), it will be adopted
   by the TPerson object. When a TPerson object is destroyed, the
   associated role objects are also destroyed.
*/
const short MAX_ROLES = 6; // Just an arbitrary limit

class TPerson {
     public:
          TPerson (const char theName[], unsigned long theSSN,
                unsigned long theBirthDate, const char theAddress[]);
          TPerson (const TPerson& source);
```

```
          TPerson& operator= (const TPerson& source);
          ~TPerson();

          // Add one more role to this person. Return true if successful,
          // false if it is not possible to add a new role
          bool AdoptNewRole(TUniversityMember* newRole);
          // Change the active role of this person
          TUniversityMember* SetActiveRole(TUniversityMember* thisOne);
          // Is this one of the roles being played by this person?
          // Returns true if successful, false otherwise
          bool IsRoleValid(const TUniversityMember* thisOne) const;
          // Return the active role of this person
          TUniversityMember* GetActiveRole() const { return _activeRole;}
          // Count of the number of roles played by this person
          unsigned short GetRoleCount() const;
          // This person no longer has this role. So remove it from the
          // list of roles available. Returns true if successful.
          bool DeActivateRole(TUniversityMember* thisOne);
     private:
          char*                    _name;
          char*                    _address;
          unsigned long            _ssn;
          unsigned long            _birthDate;
          // Any person cannot have more than MAX_ROLES roles
          // If this is a problem, a dynamic data structure such as a list
          // can be used. _allRoles is an array of pointers to
          // TUniversityMember objects. As and when new TUniversityMembers
          // are added, their addresses are placed in this array.
          TUniversityMember*       _allRoles[MAX_ROLES];
          unsigned short           _roleCount;
          TUniversityMember*       _activeRole;
};
```

Here is a part of the constructor that will initialize all the TUniversityMember slots to 0.

```
TPerson::TPerson(const char theName[], unsigned long theSSN, unsigned long
                                    theBirthDate, const char theAddress[])
{
    // other details not shown
    // To start with, there are no roles for the person
    for (int i=0; i < MAX_ROLES; i++)
          _allRoles[i] = 0; // no role set
    _roleCount = 0;
    _activeRole = 0;

}
```

Let's look at some other methods.

```
bool
TPerson::IsRoleValid(const TUniversityMember* thisOne) const
{
    /* We walk through the role array and compare (addresses) of the role
       pointers in the slot with the argument "thisOne". If they compare
```

```
       equal, then return true. One might feel that address comparison
       isn't adequate. If so, TUniversityMember can implement
       comparison operators to do more elaborate comparison. It is more
       important to understand what needs to be done. How it is done is up
       to the implementor.
      */
    for (int i =0; i < MAX_ROLES; i++)
          if (_allRoles[i] != 0 && (_allRoles[i] == thisOne))
                return true;

    return false; // there is no such role for this person
}

bool
TPerson::AdoptNewRole(TUniversityMember* newRole)
{
    /* Add a new role to the list of all roles available
     * Walk through the array of roles and look for a free slot
     * A free slot will have a 0. Store address in that slot
     */
    for(int i=0; i < MAX_ROLES; i++)
         if (_allRoles[i] == 0) {
                _allRoles[i] = newRole;
                _roleCount++;
                return true;
         }
    return false; // failed because this person already has too many roles.
}

// Change the active role of this person
TUniversityMember*
TPerson::SetActiveRole(TUniversityMember* thisOne)
{
    // Set the active role to the argument passed in.
    // Return the old active role.
    if (thisOne != 0) {
           TUniversityMember* oldRole = _activeRole;
           _activeRole = thisOne;
           return oldRole;
    }
    // An exception would be more appropriate than returning 0
    return 0;
}

bool
TPerson::DeActivateRole(TUniversityMember* thisOne)
{
    /* Remove a role from the list of roles
     * Walk through the array of roles and look for thisOne
     * Set it to 0
    */
    for(int i=0; i < MAX_ROLES; i++)
         if (_allRoles[i] == thisOne) {
                _allRoles[i] = 0;
                _roleCount--; // One less role played
                return true;
         }
```

```
    return false; // failed because thisOne is not a role played by this
                  // person
}
```

It is important to understand the significance of the SetActiveRole and GetActiveRole methods. At any time, a TPerson is capable of playing only one role. Any client of TPerson depends on these methods to access the current role. Without these methods, a client will not be able to access any TUniversityMember object contained in a TPerson because the role objects are private implementation data inside TPerson. For proper functioning of the system, every client must use the SetActiveRole and GetActiveRole methods. Access to SetActiveRole might also be controlled to ensure that only authorized clients can modify roles of a TPerson.

Here is a skeleton of the TUniversityMember class.

```
class TUniversityMember {
     public:
          // Create a role object for the person thisPerson
          // roleName is the name of the derived class that created it.
          // This would be TTeacher, TStudent, etc.
          TUniversityMember(const char role[], const TPerson* thisPerson);
          TUniversityMember(const TUniversityMember& copy);
          TUniversityMember& operator=(const TUniversityMember& assign);
          virtual ~TUniversityMember();

          // Return the person associated with this role object.
          TPerson* RoleFor() const { return _ownerOfRole; }

          // Associate this role with a different person. Returns the
          // existing TPerson associated with this role.
          TPerson* ChangeOwnerTo(TPerson* newPerson);
          EDepartment GetDepartment() const;
          void SetDepartment(EDepartment newDept);
     private:
          // roleName is the name of the derived class that created it.
          char* _roleName;
          TPerson* _ownerOfRole;
          // Information common to all roles
          EDepartment _department;
};
```

Just to show what this class does, here is the constructor.

```
TUniversityMember::TUniversityMember(const char role[],
                                     const  TPerson* thisPerson)
{
    if (role != 0) {
           _roleName = new char [strlen(role) + 1];
           strcpy(_roleName, role);
    }
    else _roleName = 0;
    _ownerOfRole = thisPerson;
}
```

The way we use TTeacher, TStudent, etc. hasn't changed. We still instantiate these classes when needed. But now, they are roles, and so every object (of TTeacher, TStudent, etc.) must be associated with the correct person. A role belongs to a person. We need to setup the association between any TUniversityMember (the role) object and the corresponding TPerson object (the person who plays the role). This could be done in two different ways. We could pass the TPerson object to the constructor of TTeacher (TStudent, TGraduateStudent, etc.), which in turn passes it to TUniversityMember constructor. Another approach is to leave this responsibility to the university management system. TUniversityMember would have a method (SetRoleFor) that accepts a TPerson object and sets up the association of the role with that person. With the latter approach, implementation of the existing classes (TTeacher, TStudent, etc.) would mostly remain as it is. But the caveat is that a role could remain unattached to any person. With the former approach, the implementation of the class TTeacher (TStudent, TGraduateStudent, etc.) must be changed but it ensures that TUniversityMember objects are always bound to persons.

Let's look at a simple function that illustrates how RTTI is useful in this example (Fig. 6-22 and code below). Here, the function EnrollStudentInCourse takes care of the steps necessary in enrolling a student in a course. This function would exist in our earlier implementations also. It would probably be a method of some other class in the university system.

```
void EnrollStudentInCourse(const TCourse& newCourse, TPerson* aStudent)
{
     TUniversityMember* role = aStudent.GetActiveRole();
     // Now we have the active role. But we aren't sure if this role
     // is really a TStudent. We can use RTTI to verify that.

     TStudent* studentPtr = dynamic_cast <TStudent*> (role);
     if (studentPtr != 0) {
          // Wonderful, the role is really a TStudent. Invoke the
          // EnrollForCourse member function on the TStudent object
          // pointed by studentPtr ...
```

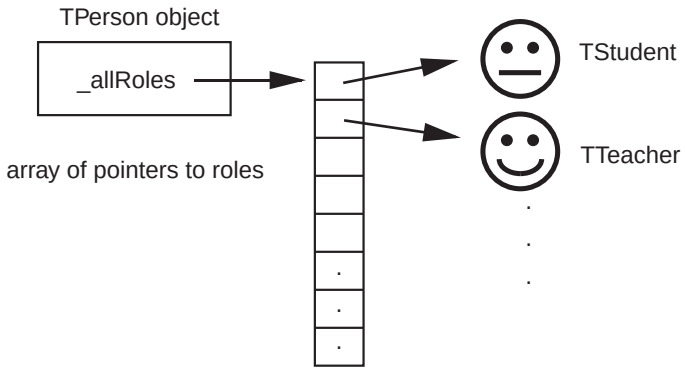


Fig. 6-22

```
            studentPtr->EnrollForCourse(newCourse);
     }
     else {
          // This role is not a TStudent. Cannot enroll person.
          // throw an exception to indicate failure
     }
     // More uninteresting code to update other information
}
```

This function could also be written as

```
    void EnrollStudentInCourse(const TCourse& newCourse,
                                        TUniversityMember* aStudent);
```

in which case the call to GetActiveRole is not needed. The caller of EnrollStudentInCourse would do the preprocessing required to access (and pass) the correct TUniversityMember object.

When we use a TUniversityMember object, we cannot do many useful things with it because it doesn't have many useful methods. A TUniversityMember is really just a unifying base class. We are more interested in determining the exact subclass of the TUniversityMember object—TStudent, TTeacher, etc. We need RTTI for such operations. Note that once we determine the exact subclass of a TUniversityMember object, the procedure to perform different chores is identical to that under the earlier design scenarios. The use of RTTI is only needed when using a TUniversityMember object.

Another Alternative for Managing Roles

In some situations, maintaining the notion of an active role may not be feasible because the relevance of the applicable role is based on state information stored inside the TPerson object. From an abstraction perspective it also makes more sense to ask the TPerson object if it is capable of playing the role, say by using a member function TPerson::GetRoleFor. This function would return a pointer to the correct role object if the person is currently capable of playing that role. Otherwise a null pointer is returned. This way, the person determines the validity of a role. It is possible to implement this scheme using RTTI.

```
class TPerson {
     // all other details as before
     TUniversityMember GetRoleFor(const type_info& whichRole) const;
};
```

Member function GetRoleFor () needs to know what role the caller is interested in. One way of doing this would be to use some character string description for every role ("Student", "Teacher", etc.). Then we can use a member function to get the role description. The problem with using character strings to identify roles is that a central registry (or some such) of role names must be used to ensure uniqueness of role names. This can be tedious and can become an unnecessary bottleneck. A more portable and language supported scheme would be to use the type_info information available with every class. The caller of GetRoleFor() would pass the type_info object of the role she is looking for.

The code in GetRoleFor() sequentially compares the type_info of the different TUniversityMember objects that are stored inside the TPerson object with the one passed to it, returning the TUniversityMember object that matches. If the person does not have any TUniversityMember object that matches the type_info argument (whichRole), then a null pointer is returned indicating the absence of any such role. Here is the skeleton code.

```
TUniversityMember*
TPerson:::GetRoleFor(const type_info& whichRole) const
{
    // Walk through the role array and compare type_info of each role with
    // whichRole.
    for(int i=0; i < MAX_ROLES; i++)
         // Compare the type_info objects
         if (typeid(*_allRoles[i]) == whichRole) // matching role found
               return _allRoles[i]; // return its address
    return 0; // role not found
}
```

A typical usage might be:

```
void EnrollStudentInCourse(const TCourse& newCourse, TPerson* person)
{
          // see if the person currently plays the role of a student?
          TUniversityMember* ptr = person->GetRoleFor(typeid(TStudent));
          if (ptr != 0) { // Yes, the person does play this role
                 // apply dynamic cast to convert to TStudent
                 TStudent* studentPtr = dynamic_cast <TStudent*> (ptr);
                 if (studentPtr != 0) {
                       // Wonderful, the role is really a TStudent. Invoke
                       // the EnrollForCourse member function on the
                       // TStudent object pointed by studentPtr ..
                       studentPtr ->EnrollForCourse(newCourse);
                 }
                 else {
                 // This role is not a TStudent. Cannot enroll .throw an
                 // exception to indicate failure
                 }
          }
}
```

With this scheme, the responsibility to determine if the person plays a role rests (appropriately) with the TPerson class. As with the earlier scheme, if the person plays more than one role of the same type (say two student roles), then locating and distinguishing between them is not an easy task.

Polymorphic Usage of TUniversityMember Objects

When a TUniversityMember object (a TUniversityMember reference or pointer) is an argument of a function, any object of the derived classes of TUniversityMember can be substituted. But the function wouldn't be able to do anything useful with the TUniversityMember object because it lacks any useful interface. Again, we need to use RTTI to de-

termine the exact subtype of the TUniversityMember object and perform necessary operations. In other words, polymorphic usage of TUniversityMember objects doesn't help much because we still need to know the exact type of the object passed in as the actual argument. True polymorphism, which we discussed elsewhere, insulated the user from the exact type of the object by allowing us to use the common interface. But in case of TUniversityMember, there is hardly any common interface. TUniversityMember just cannot capture all the interface of the derived classes (it will be futile to even attempt such a task). But since TTeacher, TStudent, etc. are no longer derived classes of TPerson, there is no need for polymorphism with TPerson. A person is just a person—nothing else (unless of course if we create new subclasses of TPerson). And one can get to the roles played by a person using the interface in the TPerson object. In essence, we have not lost any of the advantages we had in the earlier designs, but this new solution is a little more complicated because we must use RTTI with TUniversityMember objects. We have gained more flexibility at the cost of a little complexity. Now changing roles of a person is very easy.

Changes Required to the Existing Classes

To use this new design, many of the classes that we built earlier must be modified. Classes TStudent, TTeacher, TGraduateStudent, and, TGradTeachingAsst no longer inherit from TPerson (but they inherit from TUniversityMember). Moreover, TPerson is not a virtual base class anymore. That's a big relief because it eliminates all the complicated rules about most derived class, constructor calls, and name conflicts. All the methods in any class other than TPerson that need access to the name, address, etc. of TPerson must get them indirectly using the RoleFor method of TUniversityMember. For example, here is the code for TStudent::Print with this change:

```
void
TStudent::Print() const
{
    // Get the person object associated with this TStudent role object.
    // RoleFor is a method in the base class TUniversityMember.
    TPerson* person = RoleFor();
    //Ensure that the TPerson is not zero
    if (person != 0) {
          cout << "Name: " << person->GetName() << endl;
          cout << "Address: " << person->GetAddress() << endl;
    }
    cout << "This person is a " << statusLabels[(int)status]
         << " student in the department of "
         << departments[(int) department] << endl;
}
```

This new design also eliminates the need for a separate TGradTeachingAsst class. A person (a graduate student) plays the role of a teacher in the department where she works as a graduate teaching assistant. She also plays the role of a graduate student, but not both at the same time. When she is performing the role of a graduate teaching assistant, the active role would be set to the TTeacher role, and when she is performing the role of a normal graduate student, the active role (set by SetActiveRole) is TGraduateStudent. This

simplicity of multiple roles makes this design more elegant than any of our previous designs.

Creating, copying, and deleting of the role objects can be minimized very easily. Every TUniversityMember object is decoupled from the person performing that role. A TUniversityMember object contains information specific to that role. If two people are enrolled in the same set of courses and do exactly identical duties, then these TPerson objects do not differ in their roles but only in the TPerson information. The TUniversityMember objects associated with the two TPerson objects would look identical. This facilitates sharing of TUniversityMember objects. An implementation could create a single TUniversityMember (for example, TResearchAsst) object and attach it to different TPerson objects without really keeping multiple copies of them around. For example, a set of people (TPerson) might be performing research in the same area under the same professor. All these people are performing the same role but they have different name, address, etc. A single TResearchAsst object can be used to represent the roles of all these people, and this object can be attached to all TPerson objects. **[***There is one problem here. As stated earlier, every role belongs to a person. When the method RoleFor is applied on a TUniversityMember object, it returns the associated TPerson object. This is a one-to-one relationship. Every TUniversityMember object is owned by exactly one TPerson object. But if we allow the sharing of role objects (i.e., TUniversityMember objects), then our assumption about every role belonging to only one person collapses. If minimization of objects created is the goal, then we can make a group of TPerson objects share a single TUniversityMember object, but the TUniversityMember object would belong to only one TPerson at a time. The RoleFor member function returns the associated TPerson. The association can be changed dynamically by using the ChangeOwnerTo method.***]** And when this group of people are no longer involved in the research project, it might be beneficial to just keep the TResearchAsst object around for future use because another group of researchers might take the place of those who left. This avoids repeated object creation and deletion of TResearchAsst objects. The salient point here is that different people take on the same role(s). The duties (or characteristics) of a role remain the same but different people are associated with them. The roles essentially stay the same but the people playing the roles come and go. As another example, if a teacher assigned to a course is replaced by another teacher, only the TPerson associated with the TTeacher role must change. The TTeacher role object remains the same. If two teachers are assigned to the same class, there needs to be only one TTeacher role object but two TPerson objects associated with this TTeacher object. Such sharing is almost impossible with the previous designs because the person and the role are the same object. In summary, the role objects are transferable. The role is like a rental car. The car can be rented out to different customers (even jointly, where possible). The characteristics of the car never change, but the customers renting it come and go, and the same person can rent multiple vehicles (car, boat, cycle, etc.).

Most of the code implemented in our classes is still usable with minor modifications. The complete implementation of the hierarchy is left as an exercise to the reader.

Mixin Classes vs. Role Objects—Applicability

At this point, one might be wondering as to where mixin classes are more appropriate, compared with role playing objects (or vice-versa). Both mixin classes and role objects

seem attractive. In particular, one should understand where one is preferred over the other. If you are thinking along these lines, you are not alone.

Mixin classes add *static* capabilities. The decision to use (or not use) a mixin needs to be made when designing the class lattice (i.e., hierarchy). An object cannot acquire any new capabilities at run-time because the inheritance hierarchy of a class (object) cannot be changed. Once an object is created, it can only respond to messages that its class contains, including those inherited from other classes. No new feature can be added to the object at run-time. Therefore, mixin classes are useful only in situations where the decisions can be made at class hierarchy design time. Once the hierarchy is created and objects are instantiated, nothing can be changed. But mixin classes are very easy to understand and implement. Mixin classes are not very useful when many different combinations of classes are possible. Also note that the mixin class hierarchy cannot model the situation where a person plays the same role more than once. For example, it is not possible to model a person performing research in more than one department in the same semester.

Creating many mixin classes might seem like a good solution. But mixing these mixin classes in various combinations could lead to a proliferation of classes (also known as *combinatorial explosion*). For example, one can easily visualize different combinations of a person (student+teacher, student+researcher, graduate student+researcher, etc.). Such a complex hierarchy is shown in Fig. 6-23 and 6-24.

Multiple inheritance hierarchies are difficult to understand (compared with single inheritance hierarchies). The solution is even more difficult to understand when virtual base classes are added. When we add more roles (such as TEmployee, TLaboratoryAssistant, etc.) the hierarchy gets more and more complex and very soon it becomes unmanageable. Any hierarchy with too many classes is hard to understand and different combinations of mixin classes don't make it any easier for the designers, implementors, and clients.

Role playing objects (such as the TUniversityMember with TPerson objects) are more suited for dynamically changing situations. One of the major advantages with role playing objects is the elimination of combinatorial explosion of classes encountered with mixins. One might create a TPerson object without any roles, but we are free to assign new roles to this person as and when needed. The only restriction is that the TUniversityMember derived classes be known when the program is compiled. The TUniversityMember hierarchy itself is fixed, but a TPerson can acquire (and release) any role at any time. In reality, this design is more suited for dynamically changing situations, and commercial software definitely needs such flexibility. There is absolutely no constraint on the roles a person can acquire (and relinquish). It is only limited by the available derived classes of TUniversityMember. Moreover, this design permits duplication of roles, which is required in many situations. As seen earlier, there is nothing wrong in a person being a research assistant in more than one department. And it is quite natural for a student to hold many teaching assistantships. This is very easy to model with role objects but not mixin classes (at least not without data duplication). When using role objects, it is very easy to add new roles because there is no change required in TPerson. The disadvantage with role objects is its dependency on RTTI (or some such mechanism) and the need for some extra code to retrieve and convert TUniversityMember objects. The design concept is quite easy to understand but implementation isn't that easy. But the run-time flexibility achieved justifies the extra code and complexity.

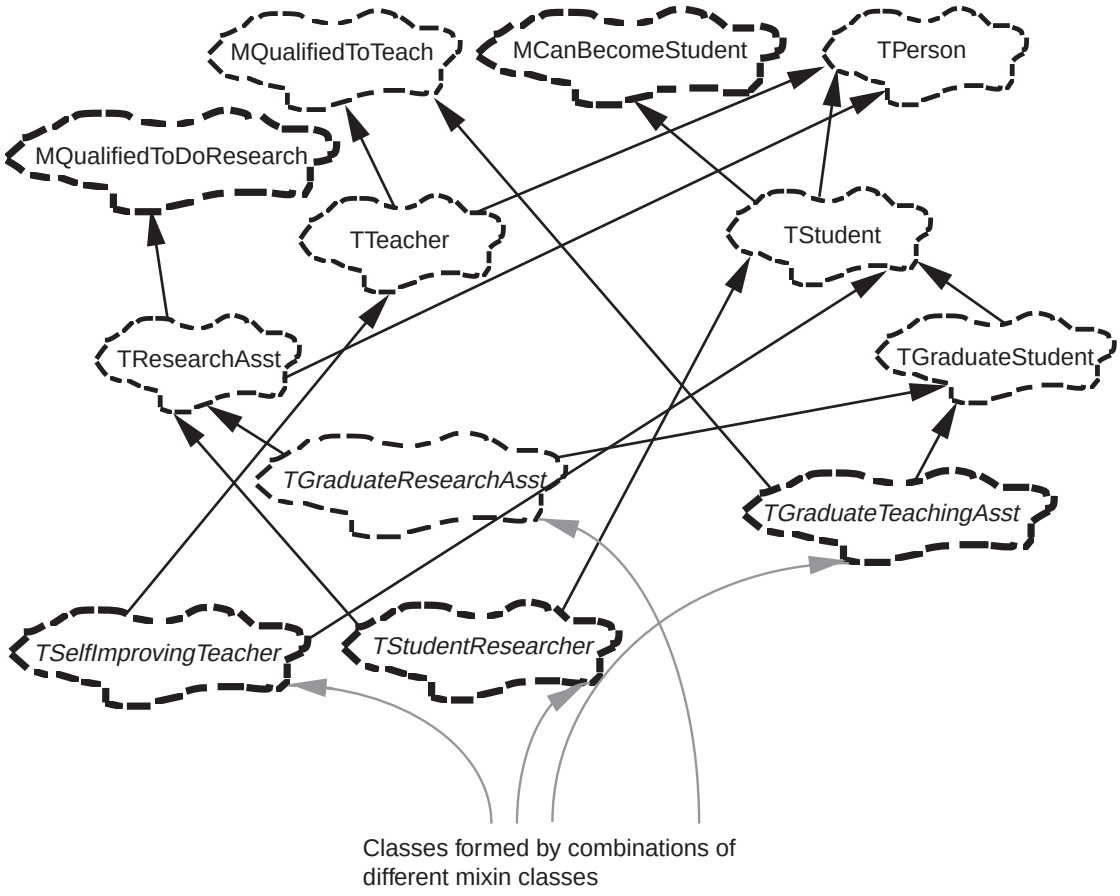


Fig. 6-23

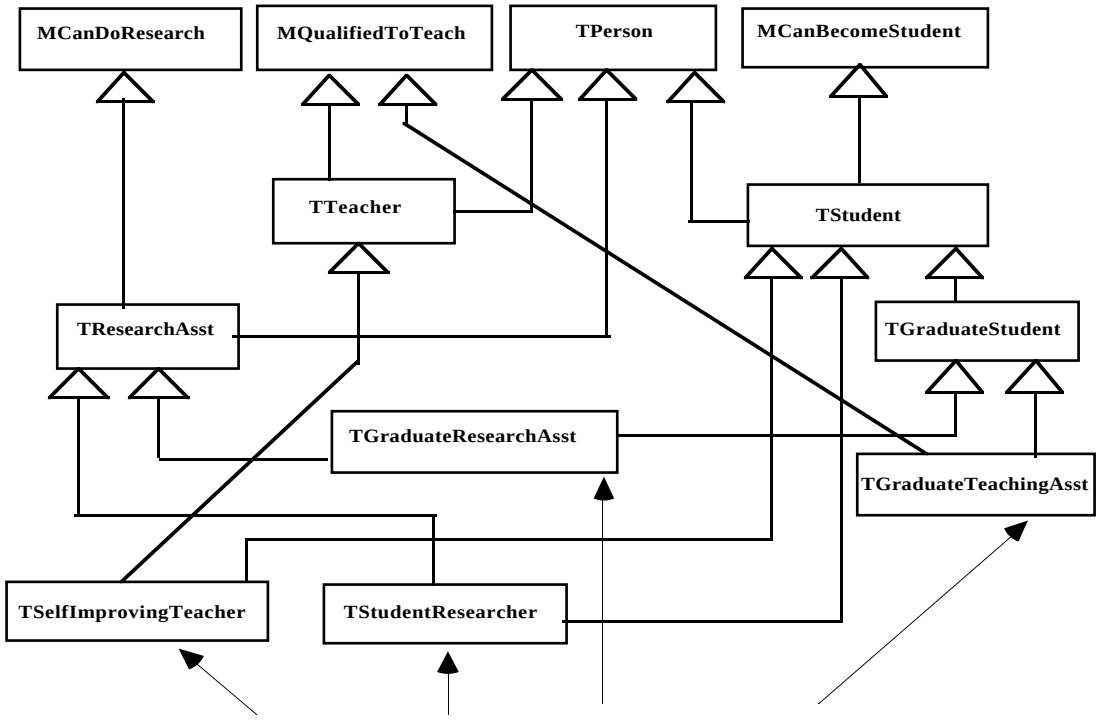
☞ **Use role playing objects if there are too many possible combinations of the roles and/or if role combinations can change dynamically.**

☞ **Use mixin classes if the number of combinations is limited and duplication of roles is not allowed.**

PRIVATE DERIVATION IN C++

Until now, we focused our attention on public inheritance, which is the most common type of class derivation. In fact, public inheritance is the only style of class derivation supported in Eiffel and Smalltalk. Under C++ we say

Combinatorial Explosion of classes with mixin classes



Classes formed by combinations of different mixin classes

Fig. 6-24

```
class TStudent : public TPerson {
     // all the details
};
```

The public keyword indicates the normal style of inheritance, which supports polymorphism. We can also write

```
class TStudent : private TPerson {
     // all the details
};
```

This indicates private derivation in C++. Let's use the phrase *private derivation* to indicate a private base class.[10] The term *inheritance* has already been used to indicate normal public inheritance, the *is-a* relationship.

[10]C++ also supports *protected* derivation wherein the base class is a protected base.

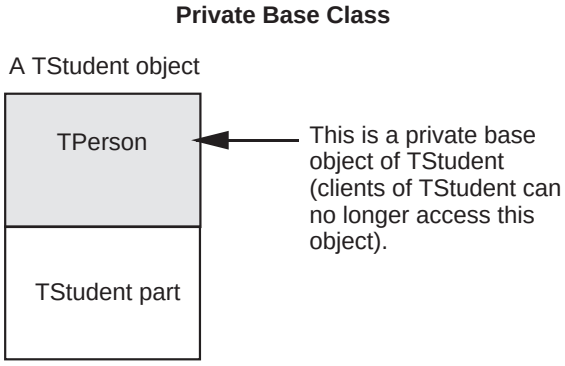


Fig. 6-25

With this declaration, class TPerson is a private base class of the derived class TStudent. A normal client of TStudent (anyone who instantiates an object of TStudent) cannot access the TPerson base class object within the TStudent object because within TStudent the base class TPerson is private. All members of the base class (private, protected, and public) TPerson become private members of the derived class TStudent and these private members cannot be accessed by clients of TStudent. Private derivation also voids the polymorphic conversion of a TStudent object to a TPerson object. One can convert from a derived class to a base class only when the base class is accessible. But with private derivation the base class object within the derived class object is no longer accessible by clients of the derived class. Hence, a TStudent object cannot be passed as the actual argument where a TPerson is expected because the called function is not allowed access to the private base class object within TStudent.

With this implementation, polymorphism of TPerson with TStudent is no longer applicable and polymorphic substitution principle is also violated. It is natural to wonder as to what new benefits are achieved by this style of derivation. We will look at the benefits very soon.

With normal (public) inheritance, a derived class inherits the interface and implementation of its base class. The interface (any member function) of the base class can be used with an object of the derived class. Moreover, every derived class also contains the complete structure (implementation) of the base class. With private derivation, the interface of the base is no longer applicable to a derived class object. But the derived class object still inherits the structure and implementation of the base class object. Hence, this style of derivation is also called *Implementation Inheritance.* The figure above (Fig. 6-25) clearly shows the structure of the derived class object. *Any member function of the derived class can still access the available methods of the base class*.[11] Polymorphism also works *within* the member functions of the derived class. Any member function of the derived class can still pass a derived class object to a base class method polymorphically. For example, if one of the member functions of TPerson expects a TPerson object (reference or pointer), any member function of TStudent can pass a TStudent object as the actual argument. Polymorphic substitution principle still works within the derived class

[11]Needless to point out that the derived class cannot access the private members of the base class.

implementation but not outside, (i.e., clients who create objects of the derived class cannot use the derived class object in place of a base class object polymorphically because the base class is not public). The derived class implementation is free to use the implementation of the base class, but clients of the derived class have no access to the member functions of the base class using objects of the derived class. Don't get confused about this access restriction with TPerson. Access to the TPerson object within a TStudent is restricted because TPerson is a private base. Anyone can still create independent TPerson objects and use them as they like. Stated differently, for normal clients, TStudent is no longer a TPerson.

The code below illustrates the scenario. Assume that TStudent has been implemented using TPerson as a private base class.

```
void f(TPerson&     nomad) // Accepts any TPerson object, polymorphically
{
          // do something with nomad object
}
main()
{
    // Michael Faraday, famous for inventions in eletro-magnetism
    TPerson faraday("Michael Faraday", 123456789, "9-22-1791",
                                          "London, United Kingdom" );
    TStudent bugs ("Bugs Bunny", 423435063, "12-1-1940", "Looney Tunes",
                                                  eArts, eFullTime);

    f(faraday); // This line compiles
    f(bugs); // Generates a compile error
    // bugs (type TStudent) cannot be converted to TPerson object because
    // TPerson is a private base class of TStudent. An ordinary client
    // cannot access the private base class object of TStudent
}
```

The implementation of Print within TStudent can still invoke Print in TPerson.

```
void
TStudent::Print() const
{
    // Ask base class to print information about the person
    TPerson::Print(); // TStudent can still access all public and protected
    // members (functions and data) of TPerson
    // print the student information
}
```

If you are still confused about the private derivation concept, read on. The next example clearly illustrates the concept.

When to Use Private Derivation

Private derivation is useful in some design scenarios. This example illustrates the concept clearly. Assume that we already have a class TList with the following interface. Just to keep it simple, only important member functions are shown. Class TListElement represents nodes on a TList.

```
class TListElement; //forward declaration

class TList {
     public:
          TList ();
          // Returns true if successful
          virtual bool Prepend(TListElement* item);
          // Returns true if successful
          virtual bool Append(TListElement* item);
          virtual void InsertAt(TListElement* what, int Where);
          virtual TListElement* DeleteItem(TListElement* which);
          virtual TListElement* RemoveFirst();
          virtual unsigned HowMany() const;
          virtual ~TList ();
          // and many more methods
     private:
          // private members used in the implementation
          TListElement* _firstElement;
};

class TListElement {
     public:
          TListElement(void* data);
          void* GetData() const { return _element;}
          // and many other methods
     private:
          void* _element;
          // and many more
};
```

When we need to implement a stack abstraction, class TStack, it would be a waste of time to write the entire implementation from scratch. A stack can be implemented using a list. But definitely stack is not a *kind of* list. Therefore, public inheritance should not be used. Class TStack can use TList as a private base class. Private derivation clearly indicates the lack of polymorphism.

```
// This is a heterogeneous stack
class TStack : private TList {
     public:
          TStack (unsigned theSize=100);
          TStack(const TStack& source);
          virtual bool Push(void* what);
          virtual void* Pop() ;
          // reuse TStack member. Use using declaration to
          // import HowMany. See Appendix for an overview of
          // namespaces and the using declaration
          using TList::HowMany;
          virtual ~TStack ();
          // and many more
     private:
          // private data members
};
```

There are some interesting points here. A stack has limited operations. The important member functions are Push (which pushes a new element onto the stack), Pop (which removes the top element from the stack and returns it), and HowMany() (which returns the count of the number of elements on the stack—depth of stack). But class TList exports many other member functions too. Definitely, we don't want clients of TStack to be able to access all the methods of TList. As is evident from the preceding paragraphs, private derivation comes to the rescue. Because TList is a private base class, clients of TStack cannot access the public member functions of TList through objects of TStack. That's really very advantageous. Unlike true inheritance, clients of TStack are limited to what TStack exports explicitly.

The member function Push in TStack is nothing but a Prepend operation on TList. So we can implement TStack::Push in terms of TListElement::Prepend. Here is the code for Push:

```
bool TStack::Push(void* theElement) {
      TList::Prepend( new TListElement(theElement) );
}
```

When Push is invoked on a TStack object, it internally invokes Prepend on its private base class TList. Nothing more to do. Applying the same logic, TStack::Pop is nothing but removal of the first element of TList, which is already supported in TList by TList::RemoveFirst.

```
void* TStack::Pop() {
     TListElement* pElem = TList::RemoveFirst();
     void* elem = pElem->GetData();
     delete pElem; // remember that we created it in Push
     return elem;
}
```

Hardly any extra code but all the functionality we need.

Re-exporting Members of the Private Base Class

The member function HowMany of TStack is a bit tricky. For the member functions Push and Pop, we had to implement them in TStack (even though they in turn call a member function of TList) because we want the appropriate names for member functions of TStack. The member function of TStack that returns the count of the number of elements is HowMany. Class TList already supports HowMany. So it would be beneficial if clients of TStack could get to TList::HowMany directly, without any intervention from TStack. In other words, the TList::HowMany must be made available to clients of TList but without any implementation in TStack. This can be done with the help of a using declaration (see Appendix for a brief overview of the namespace facility). The line

```
using TList::HowMany;
```

in the class TStack above does just that. The member function HowMany of TList is now introduced (or imported) into the namespace of the class TStack. A client of TStack can now invoke HowMany on any object of TStack. as though it were a public member of TStack. When clients of TStack invoke the method HowMany, the member function TList::HowMany is executed. In the declaration above, TStack should not specify any return type nor the argument list for the member HowMany because the declaration of HowMany in TList is used. If TList contains a set of overloaded HowMany member functions, then the declaration above inserts all such overloaded functions into the name space of class TStack (if possible).[12] Also remember that the importing class (TStack) controls the accessability of the imported name. In this example, the using declaration is in the public region of class TStack, making HowMany a public member of class TStack.

When implementing a new class, one might find that an existing class already supports all the features we need. So there is no fun in re-implementing something already in existence. But it is likely that the existing class supports many other member functions that are not applicable to our new class. For example, TList::Append is not applicable to TStack. In such situations, we want to limit the set of methods that are available to clients of the new class. Normal public inheritance does not allow the derived class to restrict access to inherited methods. Therefore, public inheritance is not applicable in such situations. This is where private derivation comes to the rescue. The derived class inherits the complete implementation of the private base class and is free to selectively re-export (with the help of the using directive) only those member functions (and even data members) that it needs. All those members of the private base class that are not re-exported by the derived class are not accessible by clients of the derived class. For example, clients of TStack do not have access to methods Append, InsertAt, etc., which are not re-exported by TStack. Class TStack only re-exports the method TList::HowMany. This is a way of reusing an implementation of an existing class but with a diminished interface. Re-exporting of members with the using declaration is not limited to member functions; even data members can be imported with a using declaration. But a derived class can only re-export those members of the private base class that it can access. Class TStack cannot import the private member TList::firstElement because TStack is not allowed access to any private member of the base class TList.

It should be clear from the preceding paragraphs that private derivation is almost always contrary to the generalization-specialization concept of public inheritance. Public inheritance is the common mode of inheritance and private derivation is useful only in some special situations. Eiffel and Smalltalk support public inheritance only.

☞ **Private derivation is not true inheritance. The is-a relation is not applicable with private derivation.**

[12]It is not possible to selectively re-export some overloaded functions among a set of overloaded functions. It is an all or nothing situation. Further, if the specified access control is not applicable to one of the overloaded functions, then the re-export fails.

Alternative to Private Derivation—Containment

Revisiting TStack, it is easy to argue that TStack can be implemented by using an object of TList internally (TStack *has-a* TList in the implementation). The code would be:

```
class TStack {
     public:
          TStack(unsigned theSize=100);
          TStack(const TStack& source); // copy ctor
          virtual bool Push(void* what);
          virtual void* Pop() ;
          virtual unsigned HowMany() const;
          ~TStack();
          // and many more
     private:
          TList _list; // TList object used in the implementation
};
```

The method Pop is

```
void*
TStack::Pop()
{
    TListElement* pElem = _list.RemoveFirst();
    void* elem = pElem->GetData();
    delete pElem; // remember that we created it in Push
    return elem;
}

and Push is
bool
TStack::Push(void* element)
{ return _list.Prepend(new TListElement(element) ); }
```

This implementation using *containment* (aggregation) is very similar to what was done with private derivation. The difference is that every method of TStack internally invokes corresponding methods on the internal object _list. But unlike private derivation, we cannot re-export TList::HowMany. It should be implemented in TStack as follows:

```
unsigned
TStack::HowMany() const { return _list.HowMany(); }
```

This is one benefit that is lost when using an object of TList internally. Private derivation allows for easy re-export of members with the help of the using declaration.. When using internal implementation objects, the enclosing class (TStack) must implement the member function using the enclosed object (_list). This is not a big drawback because we can make the call to _list.HowMany in TStack an inline member function and thus eliminate the extra function call. With this modification, there is no perceivable difference between the implementation using private derivation and that using an encapsulated object. But still the advantage of overriding member functions and access to protected members can be useful with private derivation. Class TStack can override virtual member functions of

TList even in private derivation (even if it is not very useful for clients of TStack), but that is not possible with the has-a implementation. If you have to choose between private derivation and has-a relation, definitely has-a relation is preferable. The has-a relation is much easier to understand and implement. Private derivation in most situations does not offer any extra advantages (see below). Deciding between private derivation and has-a is a pure implementation issue. Choosing one over the other does not change the behavior nor the interface of the class.

☞ **Try to use aggregation (has-a relation) instead of private derivation.**

The Need for Private Derivation

There are some implementation scenarios where private derivation is a better choice. A simple example illustrates the concept. Consider the problem of implementing a simple list class, which must be a derived class of the abstract class TAbstractList, which defines the general interface of any type of list and also allows the derived class implementors the freedom to use various implementation strategies.

```
class TListElement;
class TAbstractList {
     public:
          TAbstractList ();
          virtual bool Prepend(TListElement* item) = 0;
          virtual bool Append(TListElement* item) = 0;
          virtual void InsertAt(TListElement* what, int Where) = 0;
          virtual TListElement* DeleteItem(TListElement* which) = 0;
          virtual unsigned HowMany() const = 0;
          virtual ~TAbstractList ();
          // and many more methods
};

class TListElement {
     public:
          TListElement(void* data);
          // and many other methods
          void* GetData() { return _element; }
     private:
          void* _element
          // and many more
};
```

A fast, fixed size linked list can be implemented using simple arrays. Assume that we already have a class TFixedSizeArray.

```
class TFixedSizeArray {
    public:
    // Allocates space for "size" elements. Each element is a pointer to
    // a void allowing us to store anything we want.
    TFixedSizeArray(unsigned size);
    // Store an element into the array
    virtual void Put(void* what, unsigned where);
```

```
    // Get an element from the array
    virtual void* Get(unsigned fromWhere);
    // Sets all elements in the array to 0.
    virtual void Clear();
    virtual unsigned HowMany() const;
    // and many more
    protected:
    unsigned _count; // # elements in the array
    // Copying and assignment of objects is not allowed for general
    // clients
    TFixedSizeArray(const TFixedSizeArray& copy);
    TFixedSizeArray& operator=(const TFixedSizeArray& assign);
    private:
     // more implementation details
};
```

Our class TFixedSizeList can be implemented as follows:

```
// Multiple inheritance: TAbstractList is a public base but TFixedSizeArray
// is a private base
class TFixedSizeList : public TAbstractList, private TFixedSizeArray {
     public:
          TFixedSizeList (unsigned numElement = 100);
          // Copying and assignment of list objects is allowed
          TFixedSizeList(const TFixedSizeList& copy);
          TFixedSizeList& operator=(const TFixedSizeList& assign);

          bool Prepend(TListElement* item);
          bool Append(TListElement* item);
          void InsertAt(TListElement* what, int Where);
          TListElement* DeleteItem(TListElement* which);
          unsigned HowMany() const;
          virtual TFixedSizeList ();
          // and many more methods
     private:
          // whatever needed
};
```

By any stretch of imagination, a list is not an array but a list could be implemented using an array. Definitely, public inheritance of TFixedSizeList from TFixedSizeArray is not valid. But we cannot use has-a relation because TFixedSizeList needs access to the protected members of TFixedSizeArray. And only derived classes can access protected members of a base class. This leaves us with no other option but private derivation. Class TFixedSizeList wants to allow copying and assignment between objects but TFixedSizeArray does not (it has made the copy constructor and assignment operator protected!). But copying of TFixedSizeList objects is only possible when TFixedSizeArray objects can be copied. So we need to gain access to the protected members of TFixedSizeArray. Hence, the use of private derivation.

Another scenario that might lead to private derivation involves overriding virtual functions. If the derived class needs to override a virtual function of a base class but without public inheritance, the only alternative is private derivation because only derived

classes (private or public) can override inherited virtual member functions. This situation may not be very clear because clients can never call a function of the private base class directly anyway (they always call the derived class function). Carefully analyze the situation below.

```
class X {
     public:
          virtual void f() = 0;
};

class Y {
     public:
          virtual void fY() { this->gY();} // virtual call of gY
          virtual void gY();
};

class Z : public X, private Y {
     public:
          virtual void f(); // override pure virtual function, X::f()
          virtual void gY(); // Overrides the privately inherited Y::gY
          virtual void fY() { Y::fY(); }
};

main()
{
     Z az;
     az.fY();
}
```

In this example, class Z derives privately from Y and publicly from X. Class Z wants to override Y::gY because the behavior of Y::gY is not suitable for Z. So it does override the virtual function with Z::gY. When a client invokes az.fY(), the call ends up in Y::fY because Z::fY in turn calls the private base class function Y::fY. But Y::fY internally calls gY.[13] For proper behavior, the gY() that is executed must be the one in Z because the message was originally sent to a Z object, aZ (in the main program). If Z did not override gY then the call this->gY would end up calling Y::gY which is not appropriate for Z. So Z appropriately overrides Y::gY to provide the correct implementation. The call this->gY ends up calling Z::gY because the original object is an object of the class Z.

This flexibility is not possible if we use an embedded object of Y (instead of private derivation) in Z. Here is the example with embedded Y object.

```
class X {
     public:
          virtual void f() = 0;
};

class Y {
     public:
```

[13]It is very common for one method to call another (virtual) method. This might result in the execution of a method in the derived class that originally issued the call to the base class. In fact, we shall discuss a very powerful design technique that uses this scheme in a later chapter.

```
          virtual void fY() { this->gY();} // virtual call of gY
          virtual void gY();
};

class Z : public X {
     public:
          virtual void f(); // override pure virtual
          virtual void gY(); // new method -has no relation with Y::gY
          virtual void fY() { _embeddedY.fY(); } // new method
     private:
          Y _embeddedY;
};

main()
{
     Z az;
     az.fY();
}
```

The call az.fY will invoke Z::fY which forwards the call to Y::fY. Next Y::fY calls this->gY(), but it ends up executing Y::gY because it has not been overridden. There is no way Z can override any member functions of Y. This leads to incorrect behavior. Moreover, Y::gY cannot access any data in Z because it is not even aware that it is part of a Z object. This problem cannot be solved with embedded objects. Private derivation is the correct solution. The situation is still the same even if Y had a pure virtual function. Class Z must override the inherited pure virtual function to make Z a concrete (not abstract) class. The point here is that even though external polymorphism is not possible with objects of Z (because of the private base class), internal implementation can still use dynamic binding and overriding (between derived class and private base class).

A VERY USEFUL EXAMPLE OF MIXIN CLASSES AND PRIVATE DERIVATION

Reference counting is a very useful feature that is very frequently used in commercial software. It is necessary to understand the principle of reference counting. One type of reference counting scheme has been already shown in Chapter 4 (with reference counted TString class). In this example, we look at another scheme of reference counting.

In many situations, we would like to keep track of the number of clients using a particular resource. For example, a shared library might be loaded in memory and the operating system (OS) needs to keep track of the number of processes using that shared library.[14] Here, the OS would like to keep only one copy of the shared library in memory (Fig 6-26).

[14]For those familiar with the MS-Windows (or Windows-NT) world, a shared library is a dynamic link library (DLL).

Process (or task) P1 makes a call to a function in a shared library SL and that causes the loading of SL by the OS into memory. As long as P1 is using the shared library it will remain in memory.

Now imagine what happens when another process, P2 makes a call to a function in the same shared library SL. Since the shared library SL is already loaded into the OS memory, there is no need to load another copy of SL into memory. That would be a waste of memory. The OS should remember that more than one process is using the library SL. This is what is called as the reference count or the **use-count**. At this instance, when P1 and P2 are using SL, the use count is 2 (Fig. 6-27).

When P1 exits (or indicates to the OS that it will no longer use SL), the OS will decrement the reference count by 1 but will still keep the shared library in memory because the process P2 is still using it. The decision to keep SL in memory is based on the reference count, which is still not zero (Fig. 6-28).

It is the reference count that tells the OS when the shared library needs to be removed from memory. When the reference count on SL reaches zero (0) then it is clear that there is no other process using SL and, hence, it is safe to remove it from memory.

When P2 also stops using SL, the reference count on SL will be decremented by the OS (Fig. 6-29). At this time, the reference count on SL is zero and hence it is safe to remove SL from memory, freeing up the space occupied by it.

In this entire scenario, the key feature was the reference count. Every time a new process starts using SL, the reference count needs to be incremented and as soon as a process dies, the reference count on SL should be decremented. In reality, this is done by the OS for all shared libraries used by a process. This reference counting semantics is also needed by many other resources. Moreover, the increment and decrement operations on a reference count must be *thread-safe.* Thread safety implies that even if multiple threads are trying to access the same reference count variable, it must be guaranteed that only one thread is granted access to the variable. Further, no other thread should be able to get access to the same variable until the original thread that was granted access to the variable relinquishes control of the variable. This can be implemented in many different ways (semaphores, mutex, critical sections, etc.). The simplest method is to increment (or decrement) the variable using atomic assembly language instructions. These instructions guarantee that the operation cannot be interrupted before it completes.

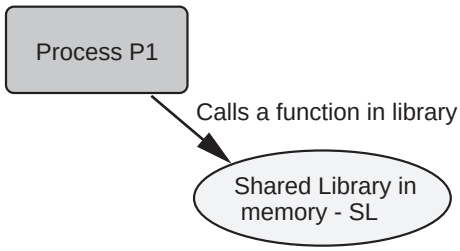


Fig. 6-26

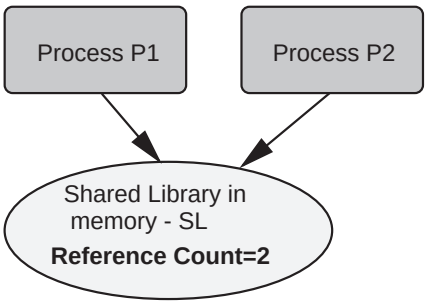


Fig. 6-27

Thus, the essence of reference counting lies in providing an integer that can be incremented and decremented safely. Any class (or resource) that requires reference counting semantics can use the reference counted variable. Is it possible to create an open scheme where any class is free to add reference counting when required? The answer is a mixin class—MReferenceCounter.

This class provides all the necessary implementation needed for reference counting. Any class requiring reference counting semantics, just mixes it in (i.e., derives from it).

```
/*
 * A thread safe reference counting mechanism.
   Methods Increment() and Decrement() guarantee thread safety. These
   methods must be implemented by the user on their platform (using
   assembly language instructions, if needed). Most clients will override
   GoingAway() to do whatever they need to do when the reference count
   drops to zero. The default implementation attempts a suicide with
   "delete this" which will cause all kinds of trouble for objects created
   on the stack - BEWARE. This class must be subclassed (inherited from).
   It cannot be instantiated.
 */
class MReferenceCounter {
     protected:
          // Constructor sets the initial count to 1
          MReferenceCounter(unsigned initialCount=1) {}

          // Increment the count by 1 and return the new count
          unsigned Increment();
```

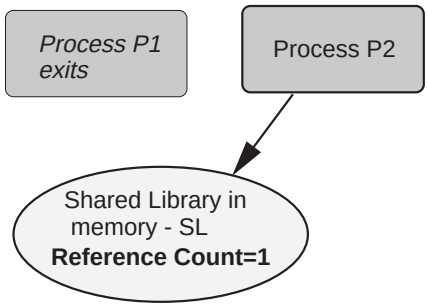


Fig. 6-28

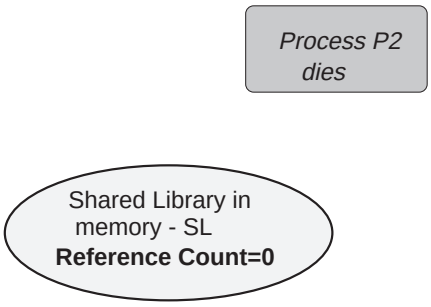


Fig. 6-29

```
          // Decrement the count by 1 and return the new count
          // If count drops to zero after the decrement operation,
          // the overridable method GoingAway() is called
          unsigned Decrement();

          // Derived classes must override this member function
          // to perform any clean-up in their implementation
          virtual void GoingAway()
          {
               // default behavior -CAREFUL
               delete this;
          }
          virtual ~MReferenceCounter() {}
     private:
          // Whatever data needed to implement the reference counting
          // mechanism. This could just be an integer with some assembly
          // language functions for the thread safe (atomic) operations. Or
          // it can use critical sections, mutex, etc. (provided by the OS)
          // to ensure thread safety.
};
```

There are some interesting things happening here.

1. This class, MReferenceCounter, is to be used only as a mixin—no direct instantiation.

2. Both Increment and Decrement methods must guarantee thread safety.

3. The most important method is GoingAway. When the reference count drops to zero after the Decrement operation, the resource (the shared library in this example) must do whatever is necessary as part of the clean-up. In the case of the shared library, the OS must reclaim the space occupied by the shared library and update any other data structures to indicate that SL is no longer available. All such operations must be done in GoingAway. For example, if the loading of shared libraries is done by the class TSharedLibraryLoader, here is how it might look. The code below is not complete by any means. It just shows some of the steps involved in managing such an operation. The emphasis is on the reference counting mechanism. Commercial implementations are much more complex.

Here is a skeleton of the shared library loader class.

```
    class TSharedLibraryLoader : private MReferenceCounter {
         public:
              TSharedLibraryLoader(const char loadThisLibrary[]);

              // Actually loads the library into memory
              // If the library has been already loaded into memory
              // the use-count (reference count) is incremented
              bool LoadLibrary();

              // Converse of LoadLibrary.
              // Decrements the reference count. If the reference count
              // drops to zero, GoingAway is automatically called
              // which will reclaim the memory used by this library
              void UnLoadLibrary();

              // Overrides the inherited function of MReferenceCounter
              virtual void GoingAway();

         private:
              char* _libName;
              bool _isLoaded;
              // Indicates if the library is already loaded into memory
    };

void
TSharedLibraryLoader::TSharedLibraryLoader(const char loadThisLibrary[])
        : MReferenceCounter(0)
        // No one is using the library yet
{
    // Just remember the name of the library
    // The actual code will not be this simple but we
    // are more interested in the reference counting semantics
    _libName = new char[strlen(loadThisLibrary)+1];
    strcpy(_libName, loadThisLibrary);
    _isLoaded = false;

}
void
TSharedLibraryLoader::LoadLibrary()
{
    if (_isLoaded)
          Increment(); // ask MReferenceCounter to add
                   // one more reference
    else {
          // Code to actually load the library into memory
          Increment(); // ask MReferenceCounter to add
                       // the first reference
    }
    return true;
}
void
TSharedLibraryLoader::UnLoadLibrary()
{
    // Very easy
    Decrement(); // Ask MReferenceCounter to remove one existing reference.
```

```
                // This might cause the
                // invocation of GoingAway which ends up
                // calling the GoingAway in our class
                // because we have overridden the inherited
                // member function
}

void
TSharedLibraryLoader::GoingAway()
{
    // Code to remove the library from memory
    // 1. Free the memory occupied by the library
    // 2. Update internal data structures to reflect the non-availability
    // of this shared library
    _isLoaded = false;
}
```

Clients of MReferenceCounter never have to worry about the intricate details involved in managing reference counts. They only worry about calling the base class methods as needed (Increment and Decrement). If everyone uses this class in their implementation, then all classes will have the same reference counting behavior.

Private derivation has been used because a TSharedLibraryLoader is *not* a MReferenceCounter. But, TSharedLibraryLoader wants to use reference counting. But at the same time, for proper behavior, the GoingAway virtual member function must be overridden in TSharedLibraryLoader. Therefore, the best alternative is private derivation.

NOTE Normal clients of TSharedLibraryLoader should not be able to access the public member functions of MReferenceCounter directly. Managing reference counts should be left to TSharedLibraryLoader class member functions. Moreover, the use of reference counting is an internal implementation issue. Therefore, even if MReferenceCounter supported some public member functions, it should not be possible to access them through TSharedLibraryLoader class. Hence, the use of private derivation.

PORTABILITY Moreover, this scheme makes our code highly portable also. Imagine what needs to be done if we need to implement this reference counting scheme on a different system (OS or processor). All that needs to be reimplemented is the mixin MReferenceCounter class that hides all the details of thread safety and other related operations.

The reference counting scheme discussed above is useful where one needs to keep track of the usage of a resource. No copying of objects is involved. We only need to know how many clients are using the resource. This scheme is very heavily used in operating system to keep track of memory pages, shared libraries, etc. There is no master-slave object scenario as discussed in the reference counting scheme presented in Chapter 4 (with the TString class). In many situations, we cannot copy an object but we can use it. In such cases, the owner of the object needs to know the number of clients using the object. This reference counting scheme is very handy in such situations.

More useful applications of private derivation can be found in subsequent chapters.

☞ **Embedded objects do not allow overriding and access to protected members. Private derivation allows overriding of member functions of the private base by the derived class and also provides access to protected members of the private base class in the derived class.**

INHERITANCE VERSUS CONTAINMENT

Now that we understand interfaces, classes, types, inheritance and also containment (aggregation), it is time we understand their implications and differences.

As mentioned elsewhere, inheritance is a static relationship. But, containment can be implemented as either a static relation (containment by value) or as a dynamic relation (containment by reference). For reuse of functionality, one has to choose between inheritance and containment.

Inheritance is straightforward to use and also easy to understand. It is directly supported by OO languages and is a fundamental paradigm of OO programming. It makes it easy for derived classes to tailor their behavior when needed by overriding member functions of the base class. And it has the benefit of polymorphism.

But inheritance brings in a permanent binding between the base class and the derived class. As seen earlier, a derived class (using public inheritance) is always a base class. There is no way to change this relationship. In other words, it is a rigid relationship determined at compile time. A child class cannot pick some other class as its parent class at run-time (this was the kind of problem we tried to solve using the role playing classes). Another problem is with base classes that also define their representation (usually data members). The derived classes must be prepared to accept the representation of the base class as it is. Or, the base class would have to expose its representation to the derived classes (happens by default in Eiffel). Exposing the representation of a base class weakens the encapsulation of the base class. Moreover, once the representation of the base class is exposed, derived classes use it directly and their implementation is tightly coupled to the implementation of the base class. This forces the derived classes to change their implementation whenever the base class representation is changed. This is highly unfortunate. This problem can be overcome by using abstract base classes without any implementation in them.

Another reuse problem with inheritance is the suitability of the base class. When inheriting from a base class, if some part of the implementation of the base class is not suitable for the derived class, then the base class must be reimplemented or the derived class must look for a more appropriate base class. This is common problem faced by designers trying to implement derived classes in new problem domains. This kind of dependency hinders true reusability and the benefit of OOP. This can be solved by using true abstract base classes that provide little or no implementation.

Containment (has-a), on other hand, affords more flexibility. This was briefly discussed towards the end of Chapter-2 and we elaborate on that here in comparison with inheritance. With containment relationship, there is flexibility to change contained objects at run-time. This was seen in the example of role playing classes where the roles played by a TPerson could be changed dynamically. Another advantage is that encapsulation boundaries are not compromised. The class (say class A) that contains objects of another class

(say class B) must use the published interface of the contained class to manipulate it. There is no other liberty available to class A with respect to class B. This truly enforces the fundamental paradigm of OOP—honoring interfaces. Using containment allows designers to combine objects in various ways to achieve end goals. Again, we could see this with role playing classes.

Designs that use containment are usually quite shallow (no deep hierarchies). With inheritance hierarchies, controlling growth is a difficult task (recall the complexity with the Teacher-Student hierarchy). And once a hierarchy keeps growing in different directions uncontrollably, it becomes a management nightmare. In designs that favor containment, there are more objects and usually fewer classes. The problem with this is that the behavior of the application depends on the relationships among many objects, whereas with inheritance the behavior is defined in one class. For example, the behavior of TPerson depends on all the role objects contained in it at any instance. In contrast, with inheritance, a TGradTeachingAsst object's behavior is always well known.

There is a well known principle behind containment. Containment favors the *buy-and-assemble* approach rather than *implement-your-own* approach. But, it is easier said than done because all the components that are required to complete the task are usually not available. With inheritance, we can create new components by extending existing ones. When containment and inheritance are combined, very difficult problems can be solved elegantly.

In any system, neither containment nor inheritance, by itself will achieve the end goal of good design with reuse. It is a combination of these two techniques that is usually more powerful. This is what we did with the role playing classes. To complete any task, different tools are required. Any tool by itself wouldn't be sufficient to solve complex problems. But, when different tools are used together, even very difficult problems can be solved easily.

SUMMARY

Multiple inheritance hierarchies are more complex, harder to design, implement, and understand than single inheritance hierarchies.

As with single inheritance, MI is a static relationship among classes. It cannot be changed at run-time.

Many difficult design problems can be easily solved by using MI hierarchies but MI is not the answer for all complex problems.

It is very hard to achieve dynamic flexibility in MI hierarchies.

Beware of virtual base classes. It is not for the weak hearted.

Many MI hierarchies can be easily reduced to single inheritance (or no inheritance) hierarchies with containment.