SESSION 23

Abstract Classes and Factoring

Session Checklist

✔ Factoring common properties into a base class

✔ Using abstract classes to hold factored information

✔ Abstract classes and dynamic typing

30 Min. To Go

So far we've seen how inheritance can be used to extend existing classes to new applications. Inheritance also affords the programmer the capability of combining common features from different classes in a process that is called *factoring*.

Factoring

To see how factoring works, let's look at the two classes used in a hypothetical banking system, `Checking` and `Savings`. These are shown graphically in Figure 23-1.

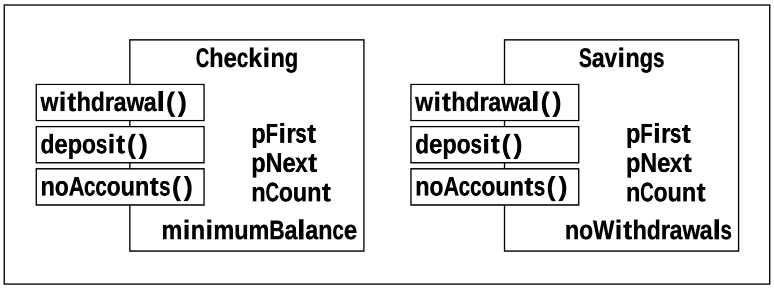

Figure 23-1
Independent classes Checking and Savings

To read this figure and the other figures that follow, remember that

- The big box is the class, with the class name at the top.
- The names in boxes are member functions.
- The names not in boxes are data members.
- The names that extend partway out of the boxes are publicly accessible members; those that do not extend outside the boxes are protected.
- A thick arrow represents the IS_A relationship.
- A thin arrow represents the HAS_A relationship.

Figure 23-1 shows that the `Checking` and `Savings` classes have a lot in common. Because they aren't identical, however, they must remain separate classes. Still, there should be a way to avoid this repetition.

We could have one of these classes inherit from the other. `Savings` has the extra members, so it makes more sense to let it inherit from `Checking`, as shown in Figure 23-2. The class is completed with the addition of data member `noWithdrawals` and the virtual overloading of member function `withdrawal()`.

Although this solution is labor saving, it's not completely satisfying. The main problem is that it misrepresents the truth. This inheritance relationship implies that a `Savings` account is a special type of `Checking` account, which is not true.

"So what?" you say, "It works and it saves effort." True, but my reservations are more than stylistic trivialities. Such misrepresentations are confusing to the programmer, both today's programmer and tomorrow's programmer. Someday, a programmer unfamiliar with this program will have to read and understand what the code is doing. Misleading tricks are difficult to reconcile and understand.

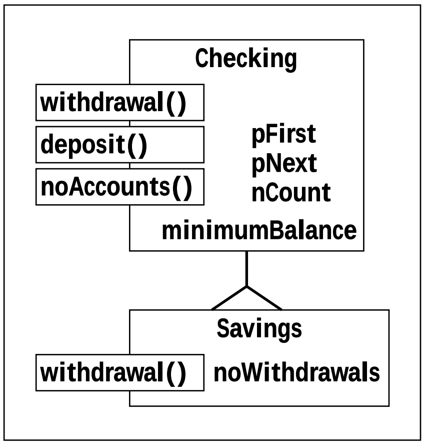

Figure 23-2
Savings implemented as a subclass of `Checking`

In addition, such misrepresentations can lead to problems down the road. Suppose, for example, that our bank changes its policies with respect to checking accounts. Let's say it decides to charge a service fee on checking accounts only if the minimum balance dips below a given value during the month.

A change like this can be easily handled with minimal changes to the class `Checking`. We'll have to add a new data member to the class `Checking`; let's call it `minimumBalance`.

But doing so creates a problem. Because `Savings` inherits from `Checking`, `Savings` gets this new data member as well. It has no use for this member because the minimum balance does not affect savings accounts. One extra data member may not be a big deal, but it does add confusion.

Changes such as this accumulate. Today it's an extra data member, tomorrow it's a changed member function. Eventually the `Savings` account class is carrying a lot of extra baggage that is applicable only to the `Checking` account class.

How do we avoid these problems? We can base both classes on a new class that is specially built for this purpose; let's call it `Account`. This class embodies all the features that a savings account and a checking account have in common, as shown in Figure 23-3.

How does this solve the problems? First, this is a more accurate description of the real world (whatever that is). In my concept of things, there really is something known as an account. Savings accounts and checking accounts are specializations of this more fundamental concept.

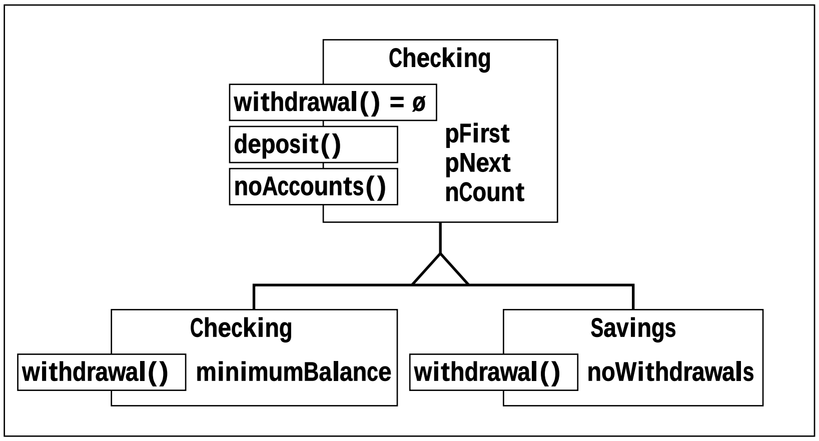

Figure 23-3
Basing Checking and Savings classes on a common Account class

In addition, the class `Savings` is insulated from changes to the class `Checking` (and vice versa). If the bank institutes a fundamental change to all accounts, we can modify `Account` and all subclasses will automatically inherit the change. But if the bank changes its policy only for checking accounts, the savings accounts remain insulated from the change.

This process of culling out common properties from similar classes is called `factoring`. This is an important feature of object-oriented languages for the reasons described so far, plus one more: reduction in redundancy.

In software, needless bulk is bad. The more code you generate, the more you have to debug. It's not worth staying up nights generating clever code just to save a few lines here or there — that type of cleverness usually boomerangs. But factoring out redundancy through inheritance can legitimately reduce the programming effort.

Tip

Factoring is legitimate only if the inheritance relationship corresponds to reality. Factoring together a class `Mouse` **and** `Joystick` **because they're both hardware pointing devices is legitimate. Factoring together a class** `Mouse` **and** `Display` **because they both make low-level operating system calls is not — the mouse and the display share no real world proprerties.**

Factoring can and usually does result in multiple levels of abstraction. For example, a program written for a more developed bank may have a class structure such as that shown in Figure 23-4.

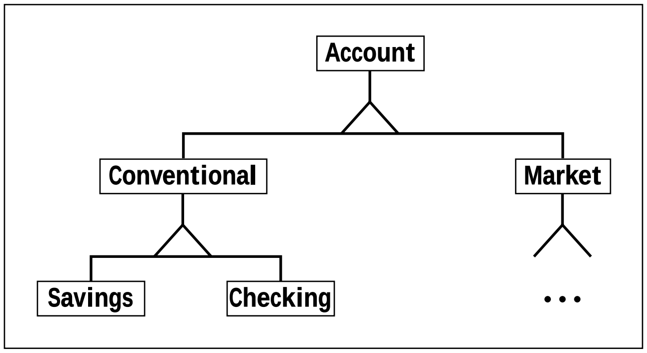

Figure 23-4
A more developed bank account hierarchy

Another class was inserted between `Checking` and `Savings` and the most general class, `Account`. This class, called `Conventional`, incorporates features common to conventional accounts. Other account types, such as stock market accounts, are also foreseen.

Such multitiered class structures are common and desirable as long as the relationships they express correspond to reality. There is, however, no one correct class hierarchy for any given set of classes.

Suppose that our bank enables account holders to access checking and stock market accounts remotely. Withdrawals from other account types can be made only at the bank. Although the class structure in Figure 23-4 seems natural, that shown in Figure 23-5 is also justifiable given this information. The programmer must decide which class structure best fits the data and leads to the cleanest, most natural implementation.

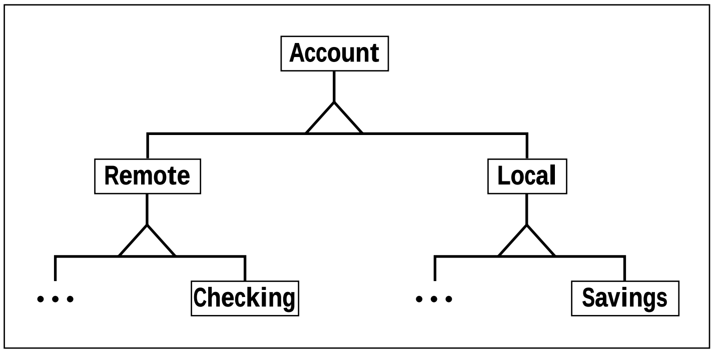

Figure 23-5
An alternate class hierarchy to that shown in Figure 23-4

20 Min.
To Go

Abstract Classes

As intellectually satisfying as factoring is, it introduces a problem of its own. Let's return one more time to the bank account classes, specifically the common base class `Account`. Let's think for a minute about how we might define the different member functions defined in `Account`.

Most `Account` member functions are no problem because both account types implement them in the same way. `withdrawal()` is different. The rules for withdrawing from a savings account are different than those for withdrawing from a checking account. Thus, we would expect `Savings::withdrawal()` to be implemented differently than `Checking::withdrawal()`. But the question is this: How do we implement `Account::withdrawal()`?

"No problem," you say. "Just go to your banker and ask, 'What are the rules for making a withdrawal from an account?' " The reply is, "What type of account? Savings or checking?" "From an account," you say, "just an account." Blank look.

The problem is that the question doesn't make sense. There's no such thing as "just an account." All accounts (in this example) are either checking accounts or savings accounts. The concept of an account is an abstract one that factors out properties common to the two concrete classes. It is incomplete, however, because it lacks the critical property `withdrawal()`. (After we get further into the details, we'll find other properties that a simple account lacks.)

Let me borrow an example from the animal kingdom. We can observe the different species of warm-blooded, baby-bearing animals and conclude that there is a concept *mammal*. We can derive classes from *mammal*, such as *canine*, *feline*, and *hominid*. It is impossible, however, to find anywhere on earth a pure mammal, that is, a mammal that isn't a member of some species. Mammal is a high-level concept that we have created — no instances of mammal exist.

Note

The concept of mammal is fundamentally different than the concept of dog. "Dog" is a name we have given to an object that exists. There is no such thing as a mammal in the real world.

We don't want the programmer to create an object of class `Account` or class `Mammal` because we wouldn't know what to do with it. To address this problem, C++ allows the programmer to declare a class that cannot be instanced with an object. The only use for such a class is that it can be inherited.

A class that cannot be instanced is called an *abstract* class.

Declaring an abstract class

An abstract class is a class with one or more pure virtual functions. A *pure virtual function* is a virtual member function that is marked as having no implementation.

A pure virtual function has no implementation because we don't know how to implement it. For example, we don't know how to perform a `withdrawal()` in class `Account`. The concept doesn't make sense. However, we can't just not include a definition of `withdrawal()` because C++ will assume that we forgot to define the function and give us a link error stating that the function is missing (and presumed forgotten).

The syntax for declaring a function pure virtual — and letting C++ know that the function has no definition — is demonstrated in the following class `Account`:

```
// Account - this class is an abstract class describing
//           the base class for all bank accounts
class Account
{
  public:
    Account(unsigned nAccNo);

    //access functions
    int accountNo();
    Account* first();
    Account* next();

    //transaction functions
    virtual void deposit(float fAmount) = 0;
    virtual void withdrawal(float fAmount) = 0;

  protected:
    //keep Accounts in a linked list so there's no limit
    static Account* pFirst;
           Account* pNext;

    // all accounts have a unique account number
    unsigned    nAccountNumber;
};
```

The = 0 after the declaration of deposit() and withdrawal() indicates that the programmer does not intend to define these functions. The declaration is a placeholder for the subclasses. The subclasses of Account are expected to overload these functions with concrete functions.

Note

A *concrete member function* is a function that is not pure virtual. All member functions prior to this Session have been concrete.

Note

Although this notation of using the =0 to indicate that a member function is abstract is bizarre, it's here to stay. There is an obscure reason, if not exactly a justification, for this notation, but it's beyond the scope of this book.

An abstract class cannot be instanced with an object; that is, you can't make an object out of an abstract class. For example, the following declaration is not legal:

```
void fn()
{
    //declare an Account with 100 dollars
    Account acnt(1234);          // this is not legal
    acnt.withdrawal(50);         // what would this do?
}
```

If the declaration were allowed, the resulting object would be incomplete, lacking in some capability. For example, what should the preceding call acnt.withdrawal(50) do? There is no Account::withdrawal().

Abstract classes serve as base classes for other classes. An Account contains all the properties that we associate with a generic bank account including the capability to make a withdrawal and a deposit. It's just that we can't define how a generic account would do such a thing — that's left to the particular subclass to define. Said another way, an account is not an account unless the user can make deposits and withdrawals, even if such operations cannot be defined except in terms of particular types of accounts, such as savings and check accounts.

We can create other types of bank accounts by inheriting from Account, but they cannot be instanced with an object as long as they remain abstract.

Making a "concrete" class out of an abstract class

The subclass of an abstract class remains abstract until all pure virtual functions are overloaded. The following class `Savings` is not abstract because it overloads the pure virtual functions `deposit()` and `withdrawal()` with perfectly good definitions.

```
// Account - this class is an abstract class describing
//           the base class for all bank accounts
class Account
{
  public:
    Account(unsigned nAccNo);

    //access functions
    int accountNo();
    Account* first();
    Account* next();

    //transaction functions
    virtual void deposit(float fAmount) = 0;
    virtual void withdrawal(float fAmount) = 0;

  protected:
    //keep Accounts in a linked list so there's no limit
    static Account* pFirst;
           Account* pNext;

    // all accounts have a unique account number
    unsigned    nAccountNumber;
};

// Savings - implement the Account concept
class Savings : public Account
{
  public:
    // constructor - savings accounts are created
    //               with an initial cash balance
    Savings(unsigned nAccNo,
            float fInitialBalance)
        : Account(nAccNo)
    {
        fBalance = fInitialBalance;
    }

    // savings accounts know how to handle
    // these operations
    virtual void deposit(float fAmount);
    virtual void withdrawal(float fAmount);

  protected:
    float fBalance;
};

// deposit and withdrawal - define the standard
//              account operations for savings accounts
void Savings::deposit(float fAmount)
{
    // ...some function...
}
void Savings::withdrawal(float fAmount)
{
    // ...some function...
}
```

An object of class `Savings` knows how to perform deposits and withdrawals when called on to do so. That is, the following makes sense:

```
void fn()
{
    Savings s(1234;
    s.deposit(100.0);
}
```

Note

The class Account has a constructor despite it being abstract. All accounts are created with an ID. The concrete Savings account class passes the account ID off to the Account base class while handling the initial balance itself. This is part of our object accountability concept — the Account class owns the account number field, so leave it to the Account class to initialize that field.

10 Min.
To Go

When is a subclass abstract?

A subclass of an abstract class can remain abstract. Suppose we added an intermediate class to the account hierarchy. For example, suppose that my bank has something called a cash account.

Cash accounts are accounts in which assets are maintained as cash as opposed to securities. Both savings accounts and checking accounts are cash accounts. All deposits to cash accounts are handled the same in my bank; however, savings accounts charge after the first five withdrawals whereas checking accounts do not charge anything.

Using this definition, the class `CashAccount` can implement `deposit()`because that operation is well defined and common to all forms of cash account; however, `CashAccount` cannot implement the member function `withdrawal()` because different forms of cash account handle that operation differently.

In C++, the classes `CashAccount` and `Savings` appear as follows:

```
// CashAccount - a cash account holds all assets in cash
//               rather than securities. All cash accounts
//               require deposits in currency. All deposits
//               are handled the same. Withdrawals are
//               handled differently for different forms
//               of cash accounts
class CashAccount : public Account
{
  public:
    CashAccount(unsigned nAccNo,
                float fInitialBalance)
        : Account(nAccNo)
    {
        fBalance = fInitialBalance;
    }

    //transaction functions
    // deposit - all cash accounts accept deposits
    //           in currency
    virtual void deposit(float fAmount)
    {
        fBalance += fAmount;
    }

    // access functions
    float balance()
    {
        return fBalance;
    }

  protected:
    float fBalance;
};

// Savings - a savings account is a form of cash
//           account; the withdrawal operation is
//           well defined for savings accounts
class Savings : public CashAccount
{
  public:
    Savings(unsigned nAccNo,
            float fInitialBalance = 0.0F)
       :CashAccount(nAccNo, fInitialBalance)
    {
        // ...whatever else a savings account needs
        // to do that a cash account doesn't already...
    }

    // a savings account knows how to process
    // withdrawals as well
    virtual void withdrawal(float fAmount);
};

// fn - a test function
void fn()
{
    // open a savings account with $200 in it
```

```
        Savings savings(1234, 200);

        // deposit $100
        savings.deposit(100);

        // and withdraw $50 back out
        savings.withdrawal(50);
    }
```

The class `CashAccount` remains abstract because it overloads the `deposit()` but not the `withdrawal()` member functions. `Savings` is concrete because it overloads the final remaining pure virtual member function.

The test function `fn()` creates a `Savings` object, makes a deposit, and then makes a withdrawal.

Note

Originally, every pure virtual function in a subclass had to be overloaded, even if the function was overloaded with another pure virtual function. Eventually, people realized that this was as silly as it sounds and dropped the requirement. Neither Visual C++ nor GNU/C++ require it, but older compilers may.

Passing an abstract object to a function

Even though you can't instance an abstract class, it is possible to declare a pointer or a reference to an abstract class. With polymorphism, however, this isn't as crazy as it sounds. Consider the following code snippet:

```
void fn(Account* pAccount){    //this is legal
{
    pAccount->withdrawal(100.0);
}

void otherFn()
{
    Savings s;

     //this is legitimate because Savings IS_A Account
    fn(&s);
}
```

Here, pAccount is declared as a pointer to an Account. The function fn() is justified in calling pAccount->withdrawal() because all accounts know how to process their own withdrawals. However, it is understood that when the function is called, it will be passed the address of some nonabstract subclass object such as Savings.

It is important to note that, any object received by fn() will be of either class Savings or some other nonabstract subclass of Account. The function fn() is assured that we will never pass an actual object of class Account, because we could never create one to pass in the first place. That is, the following could never happen because C++ wouldn't allow it to happen:

```
void otherFn()
{
    // the following is not allowed because Account is
    // an abstract class
    Account a;

    fn(&a);
}
```

It is key that fn() was justified in calling withdrawal() with its abstract Account object because every concrete subclass of Account knows how to perform a withdrawal() operation.

Note

A pure virtual function represents a promise to implement some feature in its concrete subclasses.

Why are pure virtual functions needed?

If withdrawal() can't be defined in the base class Account, why not leave it out? Why not define the function in Savings and keep it out of Account? In many object-oriented languages, you can do just that. But C++ wants to be capable of checking that you really know what you're doing.

C++ is a strongly typed language. When you refer to a member function, C++ insists that you prove that the member function exists in the class you specified. This avoids unfortunate run-time surprises when a referenced member function turns out to be missing.

Let's make the following minor changes to Account to demonstrate the problem:

```
class Account
{
    //just like before but without withdrawal() declared
};
class Savings : public Account
{
  public:
    virtual void withdrawal(float fAmount);
};

void fn(Account* pAcc)
{
    //withdraw some money
    pAcc->withdrawal(100.00F);  //this call is not allowed;
                               //withdrawal() is not a member
                               //of class Account
};
int otherFn()
{
    Savings s;    //open an Account
    fn(&s);
    //...continues on...
}
```

The otherFn() function operates the same as before. Just as before, the function fn() attempts to call withdrawal() with the Account object that it receives. Because the function withdrawal() is not a member of Account, however, the compiler generates an error.

In this case, the Account class has made no commitment to implement a withdrawal() member function. There could be a concrete subclass of Account that did not define a withdrawal() operation. In this case, the call to pAcc->withdrawal() would have no place to go — this is a possibility that C++ cannot accept.

Done!

REVIEW

Dividing classes of objects into taxonomies based on decreasing similarity is known as factoring. Factoring almost inevitably leads to classes that are conceptual rather than concrete. A Human is a Primate is a Mammal; however, the class Mammal is conceptual — there is no instance of Mammal that isn't some specific species.

You saw an example of this with the class `Account`. While there are savings accounts and checking accounts there is no object that is simply an account. In `C++`, we say that the class `Account` is abstract. A class becomes abstract as soon as one of its member functions does not have a firm definition. A subclass becomes concrete when it has defined all of the properties that were left dangling in the abstract base class.

- A member function that has no implementation is called a pure virtual function. Pure virtual functions are noted with an equal sign followed by a 0 in their declarations. Pure virtual functions have no definition.
- A class that contains one or more pure virtual functions is called an abstract class.
- An abstract class cannot be instanced.
- An abstract class may act as a base class for other classes.
- Subclasses of an abstract class become concrete (that is, nonabstract) when it has overridden all of the pure virtual functions that it inherits.

QUIZ YOURSELF

1. What is factoring? (See "Factoring.")
2. What is the one differentiating characteristic of an abstract class in `C++`? (See "Declaring an Abstract Class.")
3. How do you create a concrete class out of an abstract class? (See "Making a Concrete Class out of an Abstract Class.")
4. Why is it possible to declare a function `fn(MyClass*)` if `MyClass` is abstract? (See "Passing an Abstract Object to a Function.")