ATM 案例研討，第二部分：實作物件導向設計

26

You can't work in the abstract.
—I. M. Pei
To generalize means to think.
—Georg Wilhelm Friedrich Hegel
We are all gifted.That is our inheritance.
—Ethel Waters
Let me walk through the fields of paper touching with my wand dry stems and stunted butterflies…
—Denise Levertov

學習目標

在本章中，你將學到：

- 將繼承的概念加入 ATM 的設計。
- 將多型的概念加入 ATM 的設計。
- 使用以 UML 爲基礎的物件導向設計來實作 ATM 系統。
- 詳細研讀 ATM 軟體系統的程式碼，解釋實作時碰到的問題。

本章綱要

26.1　簡介

在第 25 章中，我們以物件導向的設計方法，完成了 ATM 系統設計。現在，我們要開始用 C++ 實作先前完成的物件導向設計。在第 26.2 節，我們會示範如何把類別示意圖表轉化成 C++ 程式碼。在 26.3 節，我們將設計進一步改良，加入繼承和多型的概念。在 26.4 節，我們列出 ATM 軟體系統的完整 C++ 程式碼。這些程式碼都經過仔細的注解，針對實作的細節也有完整的討論。研讀這個應用程式，讓你有機會先接觸將來可能在業界碰到的應用程式。

26.2　開始撰寫 ATM 系統類別程式

[請注意：這一節可以接在第 9 章之後閱讀。]

可見度

本小節介紹的是類別成員的存取修飾子 (access specifier)。存取修飾子 `public` 和 `private` 可決定物件的屬性或操作對其他物件來說的**可見度 (visibility)**。開始實作程式之前，必須考慮類別中哪些屬性或操作可以開放爲 `public`，哪些必須爲 `private`。

之前我們曾觀察到資料成員通常需爲 `private`，而供外界使用的成員函式需爲 `public`。只會被其他成員函式所呼叫的成員函式稱爲工具函式 (utility functions)；這種函式應該是 `private` 的。在 UML 中，我們用**可見度記號 (visibility markers)** 表示屬性和操作的可見度。公用 (public) 的屬性或操作前面標示一個加號 (+)，而私有 (private) 的屬性或操作則標示減號 (-)。圖 26.1 顯示類別示意圖加上可見度記號的樣子。[請注意：圖 26.1 未包括任何操作參數。這是正常的；加入可見度記號不會影響我們在圖 25.18–25.21 定義的參數)。]

瀏覽性

開始用 C++ 實作設計之前，我們再介紹另一種 UML 記號。我們替圖 26.2 類別圖中的聯繫線 (association lines) 加上瀏覽箭號 (navigability arrow)，進一步細分 `ATM` 系統中類別間的關係。**瀏覽箭號** (**navigability arrows**，圖示爲具粗箭頭的箭號) 依第 25.7 節介紹的溝通示意圖 (communication diagram) 和順序示意圖 (sequence diagram) 提供的資訊，表示合作關係進行的方向。依據 UML 設計撰寫程式時，瀏覽箭號可幫助你辨別哪些物件需要參照其它的物件。例如，瀏覽箭號從 `ATM` 類別指向 `BankDatabase` 類別，表示可以由前者檢視後者，所以 `ATM` 類別可以叫用 `BankDatabase` 類別的操作。但在圖 26.2 中，沒有由 `BankDatabase` 類別指向 `ATM` 類別的箭號，所以 `BankDatabase` 不能存取 `ATM` 類別的操作。類別示意圖中有些合作關係兩邊都有箭頭，或兩邊都沒有箭頭；這兩種情形都稱作**雙向瀏覽性 (bidirectional navigability)**；意思是說，合作關係兩端的物件可以互相參照。

圖 26.2 的類別示意圖與圖 25.10 一樣，省略了 `BalanceInquiry` 和 `Deposit` 兩個類別以簡化圖示。這兩個類別所參與的聯繫關係，其瀏覽性與 `Withdrawal` 類別的差不多。請回想在第 25.3 節中，`BalanceInquiry` 與 `Screen` 類別有聯繫。我們可以由 `BalanceInquiry` 類別檢視 `Screen` 類別，但無法由 `Screen` 類別檢視 `BalanceInquiry` 類別。所以，若要在圖 26.2 中建立 `BalanceInquiry` 類別的模型，必須在此聯繫靠 `Screen` 類別的那端加上箭號。再回想 `Deposit` 類別，它和 `Screen`、`Keypad`、`DepositSlot` 等類別有聯繫。我們可從 `Deposit` 類別檢視上述類別，但反

之則不行。所以我們在 `Deposit` 類別與上述類別的聯繫線上，在 `Screen`、`Keypad`、`DepositSlot` 等類別那端加上箭號。[請注意：我們將在第 26.3 節利用物件導向的繼承概念簡化系統結構，到時會建立這些類別之間的聯繫模型。]

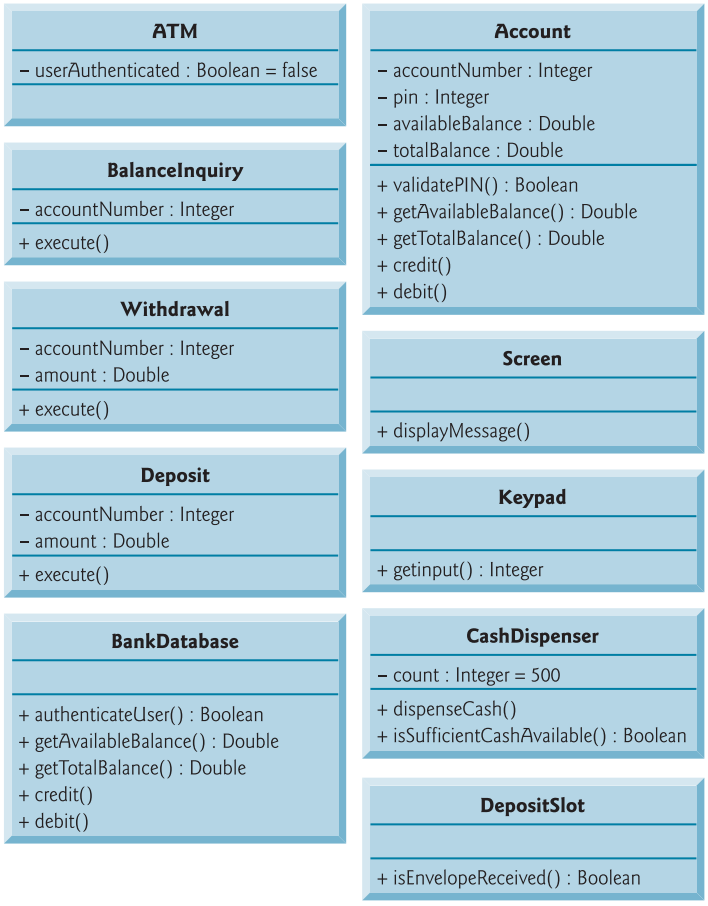

圖 26.1　包含可見度記號的類別示意圖

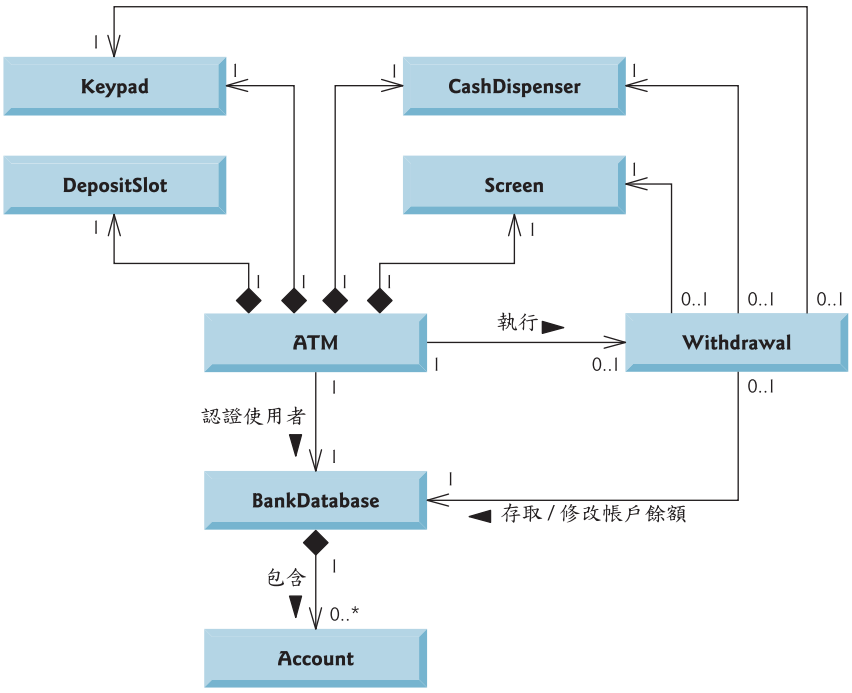

圖 26.2　包含瀏覽箭頭的類別示意圖

實作 ATM 系統的 UML 設計

現在，我們要開始實作 ATM 系統。首先，我們把圖 26.1 和圖 26.2 的圖表轉換成 C++ 標頭檔，作爲系統的骨架。在第 26.3 章中，我們會進一步修改這些標頭檔，加入物件導向的繼承觀念。26.4 節會列出完整的 C++ 程式碼。

爲舉例說明，我們首先依圖 26.1 實作類別 `Withdrawal` 的標頭檔。我們用示意圖來決定類別的各項屬性和操作，且參考圖 26.2 中的 UML 模型決定類別之間的聯繫。我們用下述五個規則實作各個類別：

1. 用類別示意圖中類別項目的第一部分作爲名稱，在標頭檔中定義類別 (圖 26.3)。透過 `#ifndef`、`#define` 和 `#endif` 這三個前置處理指令，避免在同一個原始檔中重複加入相同的標頭檔。

```
1  // Fig. 26.3: Withdrawal.h
2  // Definition of class Withdrawal that represents a withdrawal transaction
3  #ifndef WITHDRAWAL_H
4  #define WITHDRAWAL_H
5
6  class Withdrawal
7  {
8  }; // end class Withdrawal
9
10 #endif // WITHDRAWAL_H
```

圖 26.3　被前置處理包裝包圍的類別 Withdrawal 定義

2. 用類別項目第二部分的屬性宣告資料成員。例如 Withdrawal 類別中 private 的 accountNumber 屬性和 amount 屬性可寫成圖 26.4 中的程式碼。

```
1  // Fig. 26.4: Withdrawal.h
2  // Definition of class Withdrawal that represents a withdrawal transaction
3  #ifndef WITHDRAWAL_H
4  #define WITHDRAWAL_H
5
6  class Withdrawal
7  {
8  private:
9     // attributes
10    int accountNumber; // account to withdraw funds from
11    double amount; // amount to withdraw
12 }; // end class Withdrawal
13
14 #endif // WITHDRAWAL_H
```

圖 26.4　在 Withdrawal 類別標頭檔中加入屬性

3. 用類別示意圖描述的聯繫關係，宣告指向其它物件的參照或指標。例如，依圖 26.2，Withdrawal 類別可存取 Screen 類別的物件、Keypad 類別的物件、CashDispenser 類別的物件及 BankDatabase 類別的物件。類別 Withdrawal 必須維護這些物件的代表，以便傳遞訊息，故圖 26.5 第 19-22 行把這四個參照宣告爲 private 資料成員。在 26.4 節，類別 Withdrawal 的實作中可以看到，建構子將這些資料成員的初始值設成指向實際物件的參照。第 6-9 行的 #include 前置處理指令，會含括類別 Screen、Keypad、CashDispenser 和 BankDatabase 的定義，使第 19-22 行得以宣告指向這些類別物件的參照。

```
1  // Fig. 26.5: Withdrawal.h
2  // Definition of class Withdrawal that represents a withdrawal transaction
3  #ifndef WITHDRAWAL_H
4  #define WITHDRAWAL_H
5
6  #include "Screen.h" // include definition of class Screen
7  #include "Keypad.h" // include definition of class Keypad
8  #include "CashDispenser.h" // include definition of class CashDispenser
9  #include "BankDatabase.h" // include definition of class BankDatabase
10
11 class Withdrawal
12 {
13 private:
14    // attributes
15    int accountNumber; // account to withdraw funds from
16    double amount; // amount to withdraw
17
18    // references to associated objects
19    Screen &screen; // reference to ATM's screen
20    Keypad &keypad; // reference to ATM's keypad
21    CashDispenser &cashDispenser; // reference to ATM's cash dispenser
22    BankDatabase &bankDatabase; // reference to the account info database
23 }; // end class Withdrawal
24
25 #endif // WITHDRAWAL_H
```

圖 26.5　宣告指向與類別 `Withdrawal` 有聯繫的物件參照

4. 我們在圖 26.5 中含括 `Screen`、`Keypad`、`CashDispenser` 和 `BankDatabase` 的標頭檔，但程式其實不需要這麼多資訊。類別 `Withdrawal` 包含的只是指向這些類別物件的**參照**，不是眞正的物件；對編譯器來說，建立參照和建立物件所需的資訊並不相同。產生物件時，編譯器需要類別的定義，其中包含類別名稱作爲使用者自定的型別，而類別的資料成員決定該用多少記憶體儲存物件。相反的，宣告指向物件的**參照**或指標時，編譯器只需知道有這麼一個類別存在，不需知道該類別物件的實際大小。無論參照或指標所指向的物件類別爲何，它所儲存的資訊，只是實際物件的記憶體位址。表示位址所需的記憶體大小只與所用的電腦系統有關，而編譯器知道這項資訊。因此，如果只宣告指向物件的參照，不需含括其類別的完整標頭檔。我們只需提供編譯器類別的名稱，而非物件的資料儲存方式，因爲編譯器已經知道所有參照的大小了。C++ 中有一種稱爲**前置宣告 (forward declaration)** 的敘述，表示在標頭檔中包含了指向某個類別的物件或指標，但類別的定義可以在標頭檔之外。我們可把圖 26.5 中 `Withdrawal` 類別定義中的 `#include` 敘述換成 `Screen`、`Keypad`、`CashDispenser` 和 `BankDatabase` 等類別的前置宣告 (圖 26.6 第 6-9 行)；意思是說在標頭檔中，

我們在類別 `Withdrawal` 前面用這些類別的前置宣告，而不用 `#include` 完整含括它們的標頭檔。若類別 `Withdrawal` 包含實際的物件而非參照 (假設去掉第 19-22 行中的 `&` 號)，則需用 `#include` 含括完整的標頭檔。

用前置宣告代替含括完整標頭檔，能避免稱爲**循環含括 (circular include)** 的前置處理錯誤。當類別 A 含括類別 B 的標頭檔，且類別 B 含括類別 A 的標頭檔時，就會發生這種錯誤。有些前置處理器不能處理這種情形，此時會產生編譯錯誤。舉例來說，若類別 A 使用類別 B 物件的參照，則應使用前置宣告，避免循環含括的情形。

```
1  // Fig. 26.6: Withdrawal.h
2  // Definition of class Withdrawal that represents a withdrawal transaction
3  #ifndef WITHDRAWAL_H
4  #define WITHDRAWAL_H
5
6  class Screen; // forward declaration of class Screen
7  class Keypad; // forward declaration of class Keypad
8  class CashDispenser; // forward declaration of class CashDispenser
9  class BankDatabase; // forward declaration of class BankDatabase
10
11 class Withdrawal
12 {
13 private:
14    // attributes
15    int accountNumber; // account to withdraw funds from
16    double amount; // amount to withdraw
17
18    // references to associated objects
19    Screen &screen; // reference to ATM's screen
20    Keypad &keypad; // reference to ATM's keypad
21    CashDispenser &cashDispenser; // reference to ATM's cash dispenser
22    BankDatabase &bankDatabase; // reference to the account info database
23 }; // end class Withdrawal
24
25 #endif // WITHDRAWAL_H
```

圖 26.6　用前置宣告取代 `#include`

5. 按圖 26.1 類別項目的第三部分，撰寫類別成員函式的原型。若圖中未指定操作的回傳型別，則將其傳回型別宣告爲 `void`。請根據圖 6.22-6.25，宣告需要的參數。例如，我們在圖 26.7 的第 15 行撰寫 `execute` 的原型，它是類別 `WithDrawal` 的 `public` 操作，且不接收任何參數。[請注意：在 26.4 節中，我們把成員函式的定義實作放在 `.cpp` 檔之中。]

軟體工程的觀點 26.1

許多工具程式能把 UML 設計直接轉換成 C++ 程式碼，大大提高實作的效率。請造訪我們的 UML 資源中心以獲得更多有關這類「程式碼自動產生器」的資訊，網址爲 www.deitel.com/UML/。

```
1  // Fig. 26.7: Withdrawal.h
2  // Definition of class Withdrawal that represents a withdrawal transaction
3  #ifndef WITHDRAWAL_H
4  #define WITHDRAWAL_H
5
6  class Screen; // forward declaration of class Screen
7  class Keypad; // forward declaration of class Keypad
8  class CashDispenser; // forward declaration of class CashDispenser
9  class BankDatabase; // forward declaration of class BankDatabase
10
11 class Withdrawal
12 {
13 public:
14    // operations
15    void execute(); // perform the transaction
16 private:
17    // attributes
18    int accountNumber; // account to withdraw funds from
19    double amount; // amount to withdraw
20
21    // references to associated objects
22    Screen &screen; // reference to ATM's screen
23    Keypad &keypad; // reference to ATM's keypad
24    CashDispenser &cashDispenser; // reference to ATM's cash dispenser
25    BankDatabase &bankDatabase; // reference to the account info database
26 }; // end class Withdrawal
27
28 #endif // WITHDRAWAL_H
```

圖 26.7　在類別 Withdrawal 的標頭檔中加入操作

至此，讀者應已了解由 UML 圖表產生類別標頭檔的基本概念。在第 26.3 章中，我們會介紹如何進一步修改這些標頭檔，加入物件導向的繼承觀念。

26.2 節的自我測驗

26.1 說說看下面的敘述是否正確，如果不正確，請解釋其理由：類別表示圖中，某屬性前方若標記爲減號 (-) ，則該屬性無法被外界的類別直接存取。

26.2 在 圖 26.2 中， ATM 和 Screen 之間的聯繫代表：

a) 我們可以從 Screen 檢視 ATM。

b) 我們可以從 ATM 檢視 Screen。

c) a) 和 b) 都對，因爲此聯繫的瀏覽性是雙向的。

d) 以上皆非。

26.3 用 C++實作類別 `Account` 的設計。

26.3　將繼承放入 ATM 系統中

[請注意：這一節可以接在第 13 章之後閱讀。]

現在，我們重新檢視 ATM 模擬程式的設計，看看繼承能帶來什麼好處。爲了使用繼承，我們首先要找出這些類別之間的共通性。我們產生一個繼承階層，以更有效、更良好的方式模擬相似的 (不一定要相同) 類別，使我們可以多型的處理那些類別的物件，接下來，我們再修改我們的類別示意圖，納入這些新的繼承關係。最後，我們會介紹如何將剛才的設計轉換爲 C++ 的標頭檔。

在第 25.3 節當中，我們處理過金融交易系統的問題。我們當時建立了三個交易類別表達所有的交易型態；`BalanceInquiry` 類別、`Withdrawal` 類別以及 `Deposit` 類別。我們利用這些類別代表 ATM 系統執行的各種交易。圖 26.8 顯示了這些類別的屬性及操作，這些類別都含有同一項屬性 (accountNumber) 與同一項操作 (execute)。所有類別都需要 `accountNumber` 屬性，以分辨交易所處理的帳戶爲何。所有類別也都有 `execute` 操作，以供 ATM 類別執行交易時呼叫。很明顯地，`BalanceInquiry`、`Withdrawal` 以及 `Deposit` 等類別都是交易的一種。圖 26.8 顯示出這些類別的共通性，所以使用繼承的方式分離出共同特徵似乎是一種較適宜的設計方式，我們把共通的功能放在基本類別 `Transaction` 中，然後從 `Transaction` 類別衍生出類別 `BalanceInquiry`、`Withdrawal` 和 `Deposit` (圖 26.9)。

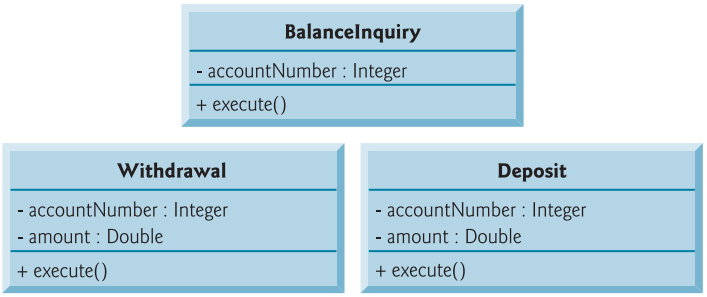

圖 26.8　`BalanceInquiry` 類別、`Withdrawal` 類別以及 `Deposit` 類別的屬性與操作

UML 提供一種稱爲**一般化 (generalization)** 的關係，來建立繼承的模型。圖 26.9 爲表示基本類別 `Transaction` 和三個衍生類別間繼承關係的類別示意圖，空心箭頭表示類別 `BalanceInquiry`、`Withdrawal` 和 `Deposit` 衍生於類別 `Transaction`。類別 `Transaction` 可以說是其衍生類別的一般化，衍生類別則可以說是 `Transaction` 類別的**特殊化 (specializations)**。

`BalanceInquiry`、`Withdrawal` 以及 `Deposit` 等類別都有整數型別的 `accountNnmber` 屬性，因此我們將這個共通屬性提出來，放入 `Transaction` 基本類別當中，便可以不需要再將 `accountNumber` 當作是衍生類別的第二部分，因爲這三個類別都可以從 `Transaction` 類別中繼承到這項屬性。然而，衍生類別並沒有辦法存取基本類別的 `private` 屬性，因此必須在 `Transaction` 類別中加入一個 `public` 的成員函式 `getAccountNumber`。每一個衍生類別都會繼承這個函式，可以使衍生類別在處理交易時存取其 `accountNumber`。

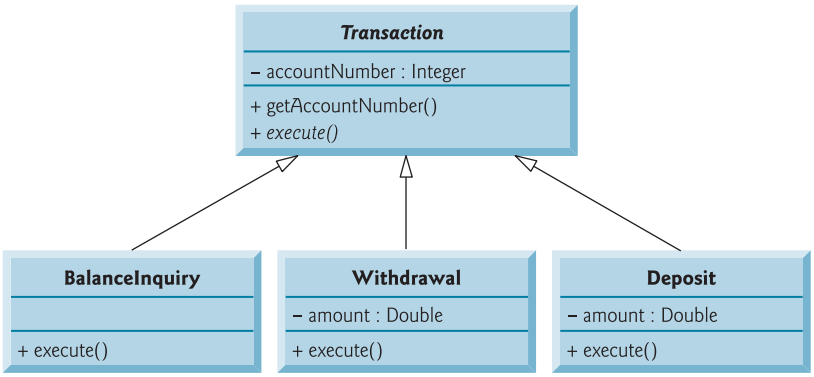

圖 26.9　顯示基本類別 `Transaction` 和衍生類別 `BalanceInquiry`、`Withdrawal` 及 `Deposit` 間一般化關係的類別示意圖

根據圖 26.8，`BalanceInquiry`、`Withdrawal` 和 `Deposit` 類別也共享了 `execute` 操作，所以 `Transaction` 類別也必須包含 `public` 的成員函式 `execute`。但是在 `Transaction` 類別中實作 `execute` 方法是沒有意義的，因爲這個方法的內容得視實際交易類型而定。因此我們將 `Transaction` 類別的成員函式 `execute` 宣告爲純粹 `virtual` 函式。這將使 `Transaction` 成爲一個抽象類別，而任何一個衍生自 `Transaction` 的類別 (如 `BalanceInquiry`、`Withdrawal` 和 `Deposit`)，都必須實

作純粹 virtual 成員函式 execute，才能成爲具象類別。我們必須在 UML 中將抽象類別 (和純粹 virtual 函式，也就是 UML 中的**抽象操作(abstract operations)** 的名稱標記爲斜體，所以圖 26.9 中的 Transaction 和其抽象成員函式 execute 以斜體表示。衍生類別 BalanceInquiry、Withdrawal 和 Deposit 的 execute 操作並未以斜體表示，每個衍生類別都以適當的實作重載了基本類別 Transaction 的成員函式 execute，對各個類別來說，被重載掉的方法各有不同的具體實作，所以圖 26.9 所示的 BalanceInquiry、Withdrawal 以及 Deposit 等類別的第三層中，都含有各自的 execute 操作。

正如你在本章中學過的，衍生類別可以繼承基本類別的介面或實作。相較於爲繼承實作而設計的階層，爲繼承介面而設計的階層會將功能放在較低的層級，基本類別會指定該階層中每個類別都有的基本功能，但衍生類別本身則擁有各自對該功能的實作。ATM 系統的繼承階層也利用了上述的繼承優點，讓 ATM 系統能夠優雅地以通用的方式執行所有的交易，每一個衍生自 Transaction 的類別都繼承了部分實作細節 (如資料成員 accountNumber)，但將繼承概念加入系統中，最大的好處是可以讓系統所有的衍生類別分享同一個介面 (如，純粹 virtual 成員函式 execute)。ATM 系統可以在任何交易中使用 Transaction 指標，當 ATM 系統經由該指標呼叫 execute 時，符合該交易的 execute 函式 (也就是該衍生類別的 .cpp 檔案中，實作的函式內容) 會自動的執行。舉例來說，假設使用者選擇執行餘額查詢。ATM 系統會把 Transaction 指標指向一個 BalanceInquiry 類別的新物件，C++ 編譯器不會發生錯誤因爲 BalanceInquiry 物件是一種 Transaction 物件。當 ATM 系統使用這個指標呼叫 execute 時，BalanceInquiry 類別的 execute 函式將會被執行。

這種多型技巧讓系統變得更容易擴充。我們必定會需要產生新的交易形式 (如轉帳或帳單付款)，這時只需要產生新的 Transaction 衍生類別，並以適當的方式重載 execute 成員函式即可。我們只要對系統程式碼做小幅更動，就可以讓使用者從主選單上選取新的交易，以及讓 ATM 實體化並執行新的衍生類別物件。ATM 可以用現有的程式碼執行新型別的交易，因爲處理交易的行爲是一致的。

就像你在本章稍早曾經學過的；程式設計者並不會將 Transaction 這種抽象類別實體化爲物件。抽象類別只需替衍生類別宣告一般化的屬性和動作，Transaction 類別所定義的是交易的概念，交易有帳號、可執行。你一定覺得好奇，既然 execute 方法缺乏具體實作，爲什麼我們仍在 Transaction 類別中宣告了純粹 virtual 成員函式 execute，因爲，我們認爲概念上這個成員函式是所有交易都需要定義的動作，而技

術上，我們在 Transaction 類別中納入 execute 函式可讓 ATM 系統 (或其他類別) 得以使用 Transaction 的指標或參照，多型地呼叫每種衍生類別的重載函式。

衍生類別 BalanceInquiry、Withdrawal 和 Deposit 等都繼承了基本類別 Transaction 的 accountNumber 屬性，但 Withdrawal 和 Deposit 類別比 BalanceInquiry 類別多了 amount 屬性，Withdrawal 與 Deposit 類別須要這個額外的屬性儲存使用者欲提領或存入帳戶的金額。BalanceInquiry 類別則不需要該屬性，只需要帳號便可執行。即使上面三個類別中有兩個類別共用這項屬性，我們仍不會將這個屬性放入 Transaction 類別中。我們只把**所有**衍生類別都會用到的性質放在基本類別裡，所以衍生類別並不會繼承不必要的屬性 (或操作)。

圖 26.10 呈現我們 ATM 模型修改後的類別示意圖，其中加入了繼承概念以及 Transaction 類別。這個模型顯示出 ATM 類別與 Transaction 類別之間的聯繫，由模型中可看出，在任一時刻中，ATM 若非正在進行一項交易，就是沒有在進行交易 (亦即在任一時刻，不是有零個，就是有一個 Transaction 型別物件存在於系統中)。因爲 Withdrawal 是 Transaction 的一種形式，所以我們不需要在類別 ATM 和 Withdrawal 類別之間再多畫出一條連線，衍生類別 Withdrawal 也繼承了所有基本類別 Transaction 跟 ATM 的聯繫。衍生類別 BalanceInquiry 和 Deposit 也繼承了這些聯繫，取代了之前在 BalanceInquiry 類別及 Deposit 類別與 ATM 類別間省略的聯繫。請再次注意，跟圖 26.9 一樣，使用空心三角箭頭意指 Transaction 類別的特殊化。

我們也加入 Transaction 類別與 BankDatabase 類別之間的聯繫 (圖 26.10)。所有 Transaction 物件都需要一個指向 BankDatabase 類別的參照，以便存取與修改帳戶裡頭的資訊。每一個衍生於 Transaction 的類別也繼承了這項參照，所以我們不需要再模擬 Withdrawal 類別和 BankDatabase 類別的聯繫，Transaction 和 BankDatabase 類別的聯繫取代了先前位在 BalanceInquiry 和 Deposit 與 BankDatabase 之間的聯繫。

我們加入 Transaction 類別和 Screen 類別的聯繫，因爲所有 Transactions 都會透過 Screen 顯示輸出給使用者，每個衍生類別也都會繼承該聯繫，因此，這裡不需要再使用過去 Withdrawal 與 Screen 間的聯繫，類別 Withdrawal 還參與了與 CashDispenser 和 Keypad 之間的聯繫，然而，Keypad 只與衍生類別 Withdrawal 和 Deposit 有聯繫，而 CashDispenser 只與 Withdrawal 有聯繫，所以我們不將它放入基本類別 Transaction 中。

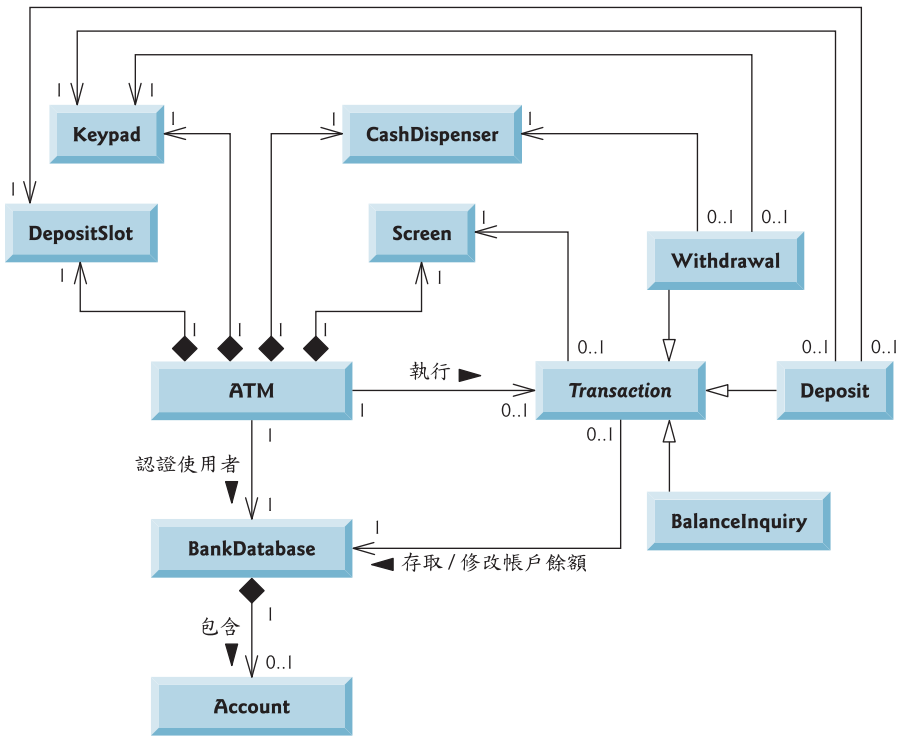

圖 26.10　ATM 系統的類別示意圖 (加入繼承概念)。請注意抽象類別的名稱 Transaction 以斜體表示

我們的類別示意圖整合了繼承 (圖 26.10) 也模擬了 `Deposit` 和 `BalanceInquiry` 的內容，我們展示出 `Deposit` 跟 `DepositSlot` 以及 `Keypad` 之間的聯繫。`BalanceInquiry` 類別並未加入其他不是繼承於 `Transaction` 類別的聯繫，因爲 `BalanceInquiry` 只會和 `BankDatabase` 及 `Screen` 有互動而已。

圖 26.1 展示了具有能見度標記的屬性跟操作。我們在圖 26.11 中展示一個修改過後的類別示意圖，其中包含抽象基本類別 `Transaction`，這個小型的圖表並沒有顯示繼承關係 (顯示於圖 26.10)，但顯示了系統中使用繼承關係後所產生的所有屬性和操作。抽象基本類別 `Transaction` 和 `Transaction` 類別中的抽象操作 `execute`，名稱都以斜體字表示。爲了節省空間，我們不再放入圖 26.10 所顯示的屬性，然而 C++ 實作會將它們包含進來。跟圖 26.1 一樣，我們也省略了所有操作的參數。但加入繼承後並不會改變我們在圖 25.18–25.21 中所顯示的參數。

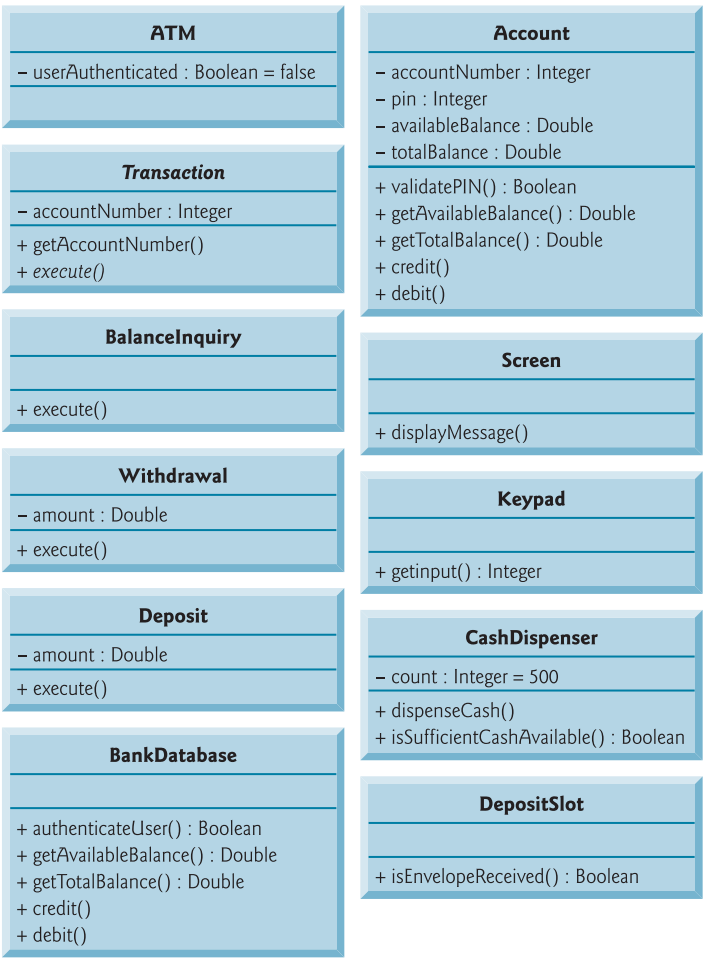

圖 26.11　加入繼承後的類別示意圖

軟體工程的觀點 26.2

完整的類別示意圖會顯示出類別之間的所有聯繫，以及每個類別的全部屬性和方法。當類別的屬性、操作和聯繫的數量龐大時 (如同圖 26.10 和圖 26.11)，我們通常會將這些資訊分成兩個類別示意圖：其中之一著重在聯繫，而另一個則著重在屬性和操作。然而研究這種類別模型圖時，很難同時考慮兩張圖表以得到這些類別的全貌。例如，我們必須參考圖 26.10 得知圖 26.11 中省略的 `Transaction` 和衍生類別之間的繼承關係。

實作這個 ATM 系統的設計 (繼承機制的體現)

現在，我們要修改之前的實作加入繼承概念，我們將以 Withdrawal 類別當作範例。

1. 如果類別 A 是類別 B 的一般化內容，則類別 B 衍生自類別 A (也就是類別 A 的特殊化版本)，例如，抽象基本類別 `Transaction` 是類別 `Withdrawal` 的一般化版本，所以，類別 `Withdrawal` 衍生自類別 `Transaction` (也是類別 `Transaction` 的特殊化版本)。圖 26.12 含有部分類別 `Withdrawal` 的標頭檔，其中類別的定義說明了 `Withdrawal` 與 `Transaction` 的繼承關係 (第 9 行)。

```
1  // Fig. 26.12: Withdrawal.h
2  // Definition of class Withdrawal that represents a withdrawal transaction
3  #ifndef WITHDRAWAL_H
4  #define WITHDRAWAL_H
5
6  #include "Transaction.h" // Transaction class definition
7
8  // class Withdrawal derives from base class Transaction
9  class Withdrawal : public Transaction
10 {
11 }; // end class Withdrawal
12
13 #endif // WITHDRAWAL_H
```

圖 26.12　衍生自 `Transaction` 類別的 `Withdrawal` 類別定義內容

2. 如果類別 A 是一個抽象類別，而類別 B 衍生於類別 A 且類別 B 是一個具象類別的話，則類別 B 必須實作類別 A 的純粹 `virtual` 函式。例如，類別 `Transaction` 含有純粹 `virtual` 函式 `execute`，所以當我們希望實作 `Withdrawal` 物件時，就必須使類別 `Withdrawal` 實作這個成員函式。圖 26.13 中含有來自圖 26.10 和 26.11 的 `Withdrawal` 類別的 C++ 標頭檔。類別 `Withdrawal` 繼承了 `Transaction` 類別的資料成員 `accountNumber`，所以 `Withdrawal` 不需要再宣告這個資料成員。同時，`Withdrawal` 也從 `Transaction` 父類別繼承了 `Screen` 類別跟 `BankDatabase` 類別的參照，所以程式碼當中也不須再加入這些參照。圖 26.11 也爲 `Withdrawal` 類別訂定了 `amount` 屬性跟 `execute` 操作。圖 26.13 的第 19 行宣告了 `amount` 資料成員。第 16 行則含有 `execute` 操作的函式原型，請回想一下，衍生類別 `Withdrawal` 爲了成爲具象類別，必須提供其基本類別 `Transaction` 中純粹 `virtual` 函式 `execute` 的實作，第 16 行的原型提醒你需要重載基本類別的純粹 `virtual` 函式，假如你要在 `.cpp` 檔案中實作這個函式，你必須提供這個原型，我們在 26.4 節中提供了實作內容。資料成員

keypad 和 cashDispenser 參照 (第 20-21 行) 衍生自 Withdrawal 類別在圖 26.10 中的聯繫。在 26.4 節的類別實作中，建構子會初始化這些實體物件的參照。再一次，爲了編譯第 20-21 行中的參照宣告，我們在第 8-9 行中放入了前置宣告。

```
1  // Fig. 26.13: Withdrawal.h
2  // Definition of class Withdrawal that represents a withdrawal transaction
3  #ifndef WITHDRAWAL_H
4  #define WITHDRAWAL_H
5
6  #include "Transaction.h" // Transaction class definition
7
8  class Keypad; // forward declaration of class Keypad
9  class CashDispenser; // forward declaration of class CashDispenser
10
11 // class Withdrawal derives from base class Transaction
12 class Withdrawal : public Transaction
13 {
14 public:
15    // member function overriding execute in base class Transaction
16    virtual void execute(); // perform the transaction
17 private:
18    // attributes
19    double amount; // amount to withdraw
20    Keypad &keypad; // reference to ATM's keypad
21    CashDispenser &cashDispenser; // reference to ATM's cash dispenser
22 }; // end class Withdrawal
23
24 #endif // WITHDRAWAL_H
```

圖 26.13　來自圖 26.10 和圖 26.11 的 Withdrawal 類別標頭檔

ATM 範例研究結論

到此，我們對 ATM 系統的物件導向設計便完成了。完整的 ATM 系統實作共 850 行程式，收錄於 26.4 節中，其中使用關鍵性的程式技術，如類別、物件、封裝、可見度、組合、繼承和多型等。程式碼中有大量的註解並遵守你學習過的設計技巧，對這個程式進行研究，對你來說將會是很棒的經驗。

26.3 節的自我測驗

26.4 UML 使用_____形式的箭號，來表示一般化關係。

a) 實心箭頭。
b) 三角形空心箭頭。
c) 菱形空心箭頭。
d) 細箭頭。

26.5 說說看下面的敘述是否正確，如果不正確，請解釋其理由：UML 規定我們要在所有的抽象類別名稱跟抽象方法名稱下頭畫底線。

26.6 請撰寫一個 C++ 標頭檔，以完成圖 26.10 和圖 26.11 中指定的 `Transaction` 類別設計，請記得放入 `Transaction` 類別中的 `private` 參照關係，並記得爲 `private` 資料成員加入 `public get` 函式，使衍生類別可以順利執行它們的任務。

26.4 ATM 案例研討實作

本節包含了第 25 章以及本章所設計之 ATM 系統的可運作完整實作，我們依照 25.3 節出現的順序來介紹這些類別：

- `ATM`
- `Screen`
- `Keypad`
- `CashDispenser`
- `DepositSlot`
- `Account`
- `BankDatabase`
- `Transaction`
- `BalanceInquiry`
- `Withdrawal`
- `Deposit`

我們遵循 26.2 節和 26.3 節的內容，以圖 26.10 和 26.11 的 UML 類別示意圖爲基礎，來撰寫這些類別的程式碼。我們依據 25.5 節的活動示意圖，以及 25.7 節的溝通示意圖與順序示意圖來建立類別的成員函式定義。請注意，我們的 ATM 設計沒有指定所有的程式邏輯，也沒有指定所有完成 ATM 程式所需要的屬性和操作。這是正常物件導向設計流程的一部分。當我們實作系統時，會依照 25.2 節的需求規格書，加入程式邏輯、屬性和行爲，以完成 ATM 系統。

最後，我們會介紹一個 C++ 程式 (`ATMCaseStudy.cpp`)，它會啓動 ATM，並示範如何使用系統中的類別。回想一下，我們開發的 ATM 系統是在個人電腦上執行的，我們利用電腦的鍵盤和螢幕來模擬 ATM 的鍵盤和螢幕。我們也只模擬 ATM 的吐鈔機和存款槽的動作。我們僅可能實作此系統，讓這些裝置在有變動時，可以不需要大幅的更動，就能夠整合入程式碼。

26.4.1　ATM 類別

ATM 類別 (圖 26.14–26.15) 代表整個 ATM 系統。圖 26.14 爲 ATM 類別定義，它被放置在 `#ifndef`、`#define` 和 `#endif` 前置處理器命令之間，以確保這些定義在程式中只會被含括一次。我們稍後會討論第 6-11 行。第 16-17 行爲此類別 `public` 成員函式的函式原型。圖 26.11 的類別示意圖並未列出任何 ATM 類別的運算，但是我們宣告 `public` 成員函式 `run` (第 17 行)，讓 `ATMCaseStudy.cpp` 的外部用戶可以對 ATM 下達執行的命令。我們也加入了預設建構子的函式原型 (第 16 行)，稍後會再做討論。

```
1  // ATM.h
2  // ATM class definition. Represents an automated teller machine.
3  #ifndef ATM_H
4  #define ATM_H
5
6  #include "Screen.h" // Screen class definition
7  #include "Keypad.h" // Keypad class definition
8  #include "CashDispenser.h" // CashDispenser class definition
9  #include "DepositSlot.h" // DepositSlot class definition
10 #include "BankDatabase.h" // BankDatabase class definition
11 class Transaction; // forward declaration of class Transaction
12
13 class ATM
14 {
15 public:
16    ATM(); // constructor initializes data members
17    void run(); // start the ATM
18 private:
19    bool userAuthenticated; // whether user is authenticated
20    int currentAccountNumber; // current user's account number
21    Screen screen; // ATM's screen
22    Keypad keypad; // ATM's keypad
23    CashDispenser cashDispenser; // ATM's cash dispenser
24    DepositSlot depositSlot; // ATM's deposit slot
25    BankDatabase bankDatabase; // account information database
26
27    // private utility functions
28    void authenticateUser(); // attempts to authenticate user
29    void performTransactions(); // performs transactions
30    int displayMainMenu() const; // displays main menu
31
32    // return object of specified Transaction derived class
33    Transaction *createTransaction( int );
34 }; // end class ATM
35
36 #endif // ATM_H
```

圖 26.14　ATM 類別定義，代表整個 ATM 系統

圖 26.14 的第 19-25 行爲 `private` 資料成員，實作類別的屬性。我們依照圖 26.10 和 26.11 的 UML 類別示意圖，建立幾乎所有的屬性 (只有一個除外)。我們將圖 26.11 中，UML 的 `Boolean` 屬性 `userAuthenticated` 實作成 C++的 `bool` 資料成員 (第 19 行)。第 20 行宣告一個不包含在 UML 設計中的資料成員 — `int` 資料成員 `currentAccountNumber`，用來紀錄目前通過驗證之使用者的帳號。我們很快會見到 ATM 類別如何利用這個資料成員。

第 21-24 行會建立一些物件，用來代表 ATM 的組件。還記得在圖 26.10 的類別示意圖中，ATM 類別和 `Screen`、`Keypad`、`CashDispenser` 以及 `DepositSlot` 類別之間具有組合關係，因此 ATM 類別必須負責產生它們。第 25 行建立 `BankDatabase`，ATM 使用它來存取和處理銀行帳戶的資訊。[請注意：假如有一個眞實的 ATM 系統，則 ATM 類別會接收一個參照，指向銀行建立的資料庫物件。然而，在這個實作中，我們只是模擬銀行的資料庫，所有 ATM 類別會建立 `BankDatabase` 物件。] 注意，第 6-10 行`#include` 類別 `Screen`、`Keypad`、`CashDispenser` 以及 `DepositSlot` 的定義，因此 ATM 可以儲存這些類別的物件。

第 28-30 以及 33 行爲 `private` 工具函式的函式原型，ATM 類別利用它們來完成工作。我們稍後會介紹這些函式如何替類別提供服務。成員函式 `createTransaction` (第 33 行) 會回傳 `Transaction` 指標。要在這個檔案中含括類別名稱 `Transaction`，我們至少需要含括類別 `Transaction` 的前置宣告 (第 11 行)。還記得吧，前置宣告會告訴編譯器這個類別存在，但是定義在別的地方。在這裡，我們使用前置宣告就足夠了，因爲我們是將 `Transaction` 指標作爲回傳型別。假如我們要建立或回傳一個眞正的 `Transaction` 物件，則我們必須 `#include` 完整的 `Transaction` 標頭檔。

ATM 類別成員函式定義

圖 26.15 包含 ATM 類別的成員函式定義。第 3-7 行 `#include` 實作檔 `ATM.cpp` 所需要的標頭檔。注意，含括 `ATM` 標頭檔可以讓編譯器確認類別的成員函式有正確地定義，也讓成員函式能夠使用類別的資料成員。

```
1  // ATM.cpp
2  // Member-function definitions for class ATM.
3  #include "ATM.h" // ATM class definition
4  #include "Transaction.h" // Transaction class definition
5  #include "BalanceInquiry.h" // BalanceInquiry class definition
6  #include "Withdrawal.h" // Withdrawal class definition
7  #include "Deposit.h" // Deposit class definition
```

圖 26.15　ATM 類別的成員函式定義

```
 8
 9  // enumeration constants represent main menu options
10  enum MenuOption { BALANCE_INQUIRY = 1, WITHDRAWAL, DEPOSIT, EXIT };
11
12  // ATM default constructor initializes data members
13  ATM::ATM()
14     : userAuthenticated ( false ), // user is not authenticated to start
15       currentAccountNumber( 0 ) // no current account number to start
16  {
17     // empty body
18  } // end ATM default constructor
19
20  // start ATM
21  void ATM::run()
22  {
23     // welcome and authenticate user; perform transactions
24     while ( true )
25     {
26        // loop while user is not yet authenticated
27        while ( !userAuthenticated )
28        {
29           screen.displayMessageLine( "\nWelcome!" );
30           authenticateUser(); // authenticate user
31        } // end while
32
33        performTransactions(); // user is now authenticated
34        userAuthenticated = false; // reset before next ATM session
35        currentAccountNumber = 0; // reset before next ATM session
36        screen.displayMessageLine( "\nThank you! Goodbye!" );
37     } // end while
38  } // end function run
39
40  // attempt to authenticate user against database
41  void ATM::authenticateUser()
42  {
43     screen.displayMessage( "\nPlease enter your account number: " );
44     int accountNumber = keypad.getInput(); // input account number
45     screen.displayMessage( "\nEnter your PIN: " ); // prompt for PIN
46     int pin = keypad.getInput(); // input PIN
47
48     // set userAuthenticated to bool value returned by database
49     userAuthenticated =
50        bankDatabase.authenticateUser( accountNumber, pin );
51
52     // check whether authentication succeeded
53     if ( userAuthenticated )
54     {
55        currentAccountNumber = accountNumber; // save user's account #
56     } // end if
57     else
58        screen.displayMessageLine(
59           "Invalid account number or PIN. Please try again." );
60  } // end function authenticateUser
```

圖 26.15　ATM 類別的成員函式定義 (續 1)

```
61
62 // display the main menu and perform transactions
63 void ATM::performTransactions()
64 {
65    // local pointer to store transaction currently being processed
66    Transaction *currentTransactionPtr;
67
68    bool userExited = false; // user has not chosen to exit
69
70    // loop while user has not chosen option to exit system
71    while ( !userExited )
72    {
73       // show main menu and get user selection
74       int mainMenuSelection = displayMainMenu();
75
76       // decide how to proceed based on user's menu selection
77       switch ( mainMenuSelection )
78       {
79          // user chose to perform one of three transaction types
80          case BALANCE_INQUIRY:
81          case WITHDRAWAL:
82          case DEPOSIT:
83             // initialize as new object of chosen type
84             currentTransactionPtr =
85                createTransaction( mainMenuSelection );
86
87             currentTransactionPtr->execute(); // execute transaction
88
89             // free the space for the dynamically allocated Transaction
90             delete currentTransactionPtr;
91
92             break;
93          case EXIT: // user chose to terminate session
94             screen.displayMessageLine( "\nExiting the system..." );
95             userExited = true; // this ATM session should end
96             break;
97          default: // user did not enter an integer from 1-4
98             screen.displayMessageLine(
99                "\nYou did not enter a valid selection. Try again." );
100            break;
101       } // end switch
102    } // end while
103 } // end function performTransactions
104
105 // display the main menu and return an input selection
106 int ATM::displayMainMenu() const
107 {
108    screen.displayMessageLine( "\nMain menu:" );
109    screen.displayMessageLine( "1 - View my balance" );
110    screen.displayMessageLine( "2 - Withdraw cash" );
111    screen.displayMessageLine( "3 - Deposit funds" );
112    screen.displayMessageLine( "4 - Exit\n" );
113    screen.displayMessage( "Enter a choice: " );
```

圖 26.15 ATM 類別的成員函式定義 (續 2)

```
114    return keypad.getInput(); // return user's selection
115 } // end function displayMainMenu
116
117 // return object of specified Transaction derived class
118 Transaction *ATM::createTransaction( int type )
119 {
120    Transaction *tempPtr; // temporary Transaction pointer
121
122    // determine which type of Transaction to create
123    switch ( type )
124    {
125       case BALANCE_INQUIRY: // create new BalanceInquiry transaction
126          tempPtr = new BalanceInquiry(
127             currentAccountNumber, screen, bankDatabase );
128          break;
129       case WITHDRAWAL: // create new Withdrawal transaction
130          tempPtr = new Withdrawal( currentAccountNumber, screen,
131             bankDatabase, keypad, cashDispenser );
132          break;
133       case DEPOSIT: // create new Deposit transaction
134          tempPtr = new Deposit( currentAccountNumber, screen,
135             bankDatabase, keypad, depositSlot );
136          break;
137    } // end switch
138
139    return tempPtr; // return the newly created object
140 } // end function createTransaction
```

圖 26.15　ATM 類別的成員函式定義 (續 3)

第 10 行宣告一個稱爲 `MenuOption` 的 `enum`，包含了對應到 ATM 主選單四個選項的常數 (餘額查詢、提款、存款以及離開)。注意，將 `BALANCE_INQUIRY` 設爲 1，會讓後續的列舉常數值分別爲 2、3、4 (列舉常數值會遞增 1)。

第 13-18 行定義 ATM 類別的建構子，它會初始化類別的資料成員。當我們建立一個 ATM 物件時，使用者尚未通過驗證，因此第 14 行會使用成員初始器將 `userAuthenticated` 設定爲 `false`。同樣地，第 15 行會將 `currentAccountNumber` 初始化爲 0，因爲現在沒有任何使用者。

ATM 成員函式 run

ATM 的成員函式 `run` (第 21-38 行) 會使用一個無限迴圈 (第 24-37 行) 重複歡迎使用者、驗證使用者，假如驗證成功會允許使用者執行交易。在通過驗證的使用者執行他想要的交易並離開之後，ATM 會重設它自己，顯示再見訊息並重新開始上述流程。我們使

用無窮迴圈來模擬 ATM 不斷執行直到銀行人員將它關掉 (使用者控制之外的行爲) 的特性。ATM 使用者可以選擇離開系統，但是無法完全關掉 ATM。

驗證使用者

在成員函式 `run` 的無窮迴圈內部，第 27-31 行會讓 ATM 重複顯示歡迎訊息，並在使用者未通過驗證時 (`!userAuthenticated` 爲 `true`)，嘗試驗證使用者。第 29 行呼叫 ATM 的 `screen` 的成員函式 `displayMessageLine`，以顯示歡迎訊息。如同案例研討中設計的 `Screen` 成員函式 `displayMessage`，成員函式 `displayMessageLine` (宣告在圖 26.16 第 13 行，定義在圖 26.17 第 20-23 行) 會向使用者顯示一個訊息，但是這個成員函式也會在顯示訊息之後輸出一個換行字元。我們在實作的過程中加入這個成員函式，讓 `Screen` 類別的客戶更能控制訊息的顯示。圖 26.15 的第 30 行會呼叫 ATM 類別的 `private` 工具函式 `authenticateUser` (第 41-60 行)，嘗試驗證使用者。

我們參考需求規格書來決定在允許交易之前，應該用哪些步驟來認證使用者。成員函式 `authenticateUser` 的第 43 行呼叫 ATM 的 `screen` 成員函式 `displayMessage`，提示使用者輸入帳號。第 44 行呼叫 ATM 的 `keypad` 成員函式 `getInput`，取得使用者的輸入，接著將使用者輸入的整數值儲存到區域變數 `accountNumber`。接著，成員函式 `authenticateUser` 會提示使用者輸入 PIN 碼 (第 45 行)，然後將使用者輸入的 PIN 碼儲存到區域變數 `pin` (第 46 行)。接下來，第 49-50 行會將使用者輸入的 `accountNumber` 和 `pin` 傳遞給 `bankDatabase` 的 `authenticateUser` 成員函式，以驗證使用者。ATM 類別會將它的 `userAuthenticated` 資料成員設定爲這個函式傳回的 `bool` 數值，假如驗證成功 (`accountNumber` 和 `pin` 與 `bankDatabase` 中的 `Account` 相符)，`userAuthenticated` 會變成 `true`，否則會保持 `false` 值。假如 `userAuthenticated` 爲 true 值，第 55 行會將使用者輸入的帳號 (`accountNumber`) 儲存在 ATM 的資料成員 `currentAccountNumber` 中。每當某個 ATM 交談期需要存取使用者帳號時，ATM 類別的其他成員函式就會使用這個變數。假如 `userAuthenticated` 爲 false 值，第 58-59 行會使用 `screen` 的 `displayMessageLine` 成員函式，指出使用者輸入了不正確的帳號或密碼，需要再試一次。注意，只有當使用者的帳號和密碼通過驗證之後，我們才會設定 `currentAccountNumber` 的值，假如資料庫無法驗證使用者，則 `currentAccountNumber` 會維持 0。

在成員函式 `run` 驗證使用者 (第 30 行) 之後，假如 `userAuthenticated` 仍然爲 `false`，則第 27-31 行的 `whil` 迴圈會再次執行。假如 `userAuthenticated` 爲 `true`，

則迴圈會停止，程式從第 33 行繼續執行，呼叫 ATM 類別的工具函式 performTransactions。

執行交易

成員函式 performTransactions (第 63-103 行) 會替通過驗證的使用者執行一個 ATM 交談期。第 66 行宣告一個區域的 Transaction 指標，指向 BalanceInquiry、Withdrawal 或 Deposit 物件，代表目前處理的 ATM 交易。我們在這裡利用 Transaction 指標來實行多型。也請注意，我們使用圖 25.7 類別示意圖中的角色名稱 currentTransaction 來替指標命名。如同我們的指標命名慣例，我們將 Ptr 附加到角色名稱的尾端，產生變數名稱 currentTransactionPtr。第 68 行宣告另一個區域變數 userExited，這是一個布林值，用來紀錄使用者是否選擇離開。這個變數控制一個 while 迴圈 (第 71-102 行)，允許使用者在選擇離開之前，可以不限次數地執行交易。在這個迴圈中，第 74 行會顯示主選單，並呼叫 ATM 的工具函式 displayMainMenu 取得使用者選擇的值 (定義在第 106-115 行)。這些成員函式會呼叫 ATM 的 screen 的成員函式，以顯示主選單，並回傳使用者透過 ATM 的鍵盤所選擇的項目。注意，這個成員函式是 const，因爲它不會修改物件的內容。第 74 行會將 displayMainMenu 回傳的使用者選項儲存到區域變數 mainMenuSelection 中。

在取得使用者選擇的主選單選項之後，成員函式 performTransactions 會使用一個 switch 敘述 (第 77-101 行) 針對這個選項做出回應。假如 mainMenuSelection 等於三個列舉常數 (代表交易型別) 之中的一個，則第 84-85 行會呼叫工具函式 createTransaction (定義在第 118-140 行)，回傳一個指標，指向所選擇的交易型別的新建物件。createTransaction 回傳的值會被設定給指標 currentTransactionPtr。第 87 行會使用 currentTransactionPtr 呼叫新物件的 execute 成員函式，以執行此交易。我們稍後會討論 Transaction 的 execute 成員函式以及三個 Transaction 的衍生類別。最後，當我們不需要 Transaction 衍生類別的物件時，第 90 行會釋放動態配置的記憶體。

我們將 Transaction 指標 currentTransactionPtr 指向三個 Transaction 衍生類別物件之一，這樣就可以用多型方式來執行交易。例如，假如使用者選擇執行餘額查詢，mainMenuSelection 等於 BALANCE_INQUIRY，則 createTransaction 會回傳指向 BalanceInquiry 物件的指標。因此，currentTransactionPtr 會指向一個 BalanceInquiry，因此呼叫 currentTransactionPtr->execute()的結果是執行 BalanceInquiry 版本的 execute。

建立交易

createTransaction 成員函式 (第 118-140 行) 會使用一個 switch 敘述 (第 123-137 行) 來實體化參數 type 指定之型別的新 Transaction 衍生類別物件。請回想，只有在 mainMenuSelection 包含的值可以對應到三個交易型別之一時，成員函式 performTransactions 才會將 mainMenuSelection 傳遞給這個成員函式。因此，type 會等於 BALANCE_INQUIRY、WITHDRAWAL 或 DEPOSIT。switch 中的每一個 case 都會讓一個暫時的 tempPtr 指向適當的 Transaction 衍生類別的新建物件。每個建構子都有它獨特的參數列，用來初始化該衍生類別物件的特定資料。BalanceInquiry 只需要目前使用者的帳號，以及 ATM 的 screen 和 bankDatabase 的參照。除了這些參數以外，Withdrawal 還需要 ATM 的 keypad 和 cashDispenser 的參照，而 Deposit 需要 ATM 的 keypad 和 depositSlot 的參照。你很快會看到 BalanceInquiry、Withdrawal 和 Deposit 的建構子各自指定參照參數，接收代表 ATM 組件的物件。因此，當成員函式 createTransaction 將 ATM 中的物件 (例如 screen 和 keypad) 傳遞給每個新建 Transaction 衍生類別物件的初始器時，新物件其實是接收了 ATM 之組合物件的**參照**。我們會在 26.4.8–26.4.11 節進一步討論這些交易類別。

離開主選單以及處理不正確的選項

在執行交易 (performTransactions 的第 87 行) 之後，userExited 仍然是 false，因此第 71-102 行的 while 迴圈會繼續重複，讓使用者回到主選單。然而，假如使用者沒有執行交易，而選擇離開，則第 95 行會將 userExited 設定爲 true，讓 while 迴圈的條件式 (!userExited) 變成 false。這個 while 迴圈是 performTransactions 成員函式的最後一個敘述，因此控制流會回到呼叫的函式 run。假如使用者輸入了不正確的主選單選項 (不是整數值 1-4)，則第 98-99 行會顯示適當的錯誤訊息，userExited 仍然爲 false，讓使用者回到主選單再試一次。

等待下一個 ATM 使用者

當 performTransactions 將控制回傳到成員函式 run，而使用者選擇離開系統時，第 34-35 行會重設 ATM 的資料成員 userAuthenticated 和 currentAccountNumber，準備服務下一個 ATM 使用者。第 36 行會顯示一個再見訊息，然後重新開始 ATM，歡迎下一位使用者。

26.4.2　Screen 類別

Screen 類別 (圖 26.16–26.17) 代表的是 ATM 的螢幕，它封裝了所有向使用者顯示輸出的相關功能。Screen 類別會以電腦螢幕來模擬眞正的 ATM 螢幕，使用 cout 以及串流插入運算子 << 來輸出文字訊息。在這個案例研討中，我們設計 Screen 類別具有一個操作 — displayMessage。爲了能更有彈性地輸出訊息到 Screen，我們現在要宣告三個 Screen 成員函式 — displayMessage、displayMessageLine 和 displayDollarAmount。這些成員函式的原型位在圖 26.16 的第 12-14 行。

```
1  // Screen.h
2  // Screen class definition. Represents the screen of the ATM.
3  #ifndef SCREEN_H
4  #define SCREEN_H
5
6  #include <string>
7  using namespace std;
8
9  class Screen
10 {
11 public:
12    void displayMessage( string ) const; // output a message
13    void displayMessageLine( string ) const; // output message with newline
14    void displayDollarAmount( double ) const; // output a dollar amount
15 }; // end class Screen
16
17 #endif // SCREEN_H
```

圖 26.16　Screen 類別定義

```
1  // Screen.cpp
2  // Member-function definitions for class Screen.
3  #include <iostream>
4  #include <iomanip>
5  #include "Screen.h" // Screen class definition
6  using namespace std;
7
8  // output a message without a newline
9  void Screen::displayMessage( string message ) const
10 {
11    cout << message;
12 } // end function displayMessage
13
14 // output a message with a newline
15 void Screen::displayMessageLine( string message ) const
16 {
17    cout << message << endl;
18 } // end function displayMessageLine
```

圖 26.17　Screen 類別的成員函式定義

```
19
20  // output a dollar amount
21  void Screen::displayDollarAmount( double amount ) const
22  {
23     cout << fixed << setprecision( 2 ) << "$" << amount;
24  } // end function displayDollarAmount
```

圖 26.17　Screen 類別的成員函式定義 (續)

Screen 類別成員函式定義

圖 26.17 包含 Screen 類別的成員函式定義。第 5 行 #include 了 Screen 類別的定義。成員函式 displayMessage (第 9–12 行) 接收一個 string 作爲引數，並使用 cout 和串流插入運算子 << 在主控台視窗將它印出來。遊標會停留在同一行，讓這個成員函式適於向使用者顯示提示訊息。成員函式 displayMessageLine (第 15–18 行) 也會列印一個 string，但是它會接著輸出一個換行符號，將遊標移到下一行。最後，成員函式 displayDollarAmount (第 21–24 行) 會以適當的格式顯示金額 (例如$123.45)。第 23 行會使用串流操作子 fixed 和 setprecision，以兩位小數輸出數值。

26.4.3　Keypad 類別

Keypad 類別 (圖 26.18–26.19) 代表的是 ATM 的小鍵盤，負責接收使用者的全部輸入。還記得我們要模擬這個硬體吧，我們使用電腦的鍵盤來模擬 ATM 的小鍵盤。電腦鍵盤包含了許多 ATM 小鍵盤上找不到的鍵。然而，我們假設使用者只會按下小鍵盤上會出現的鍵，也就是數字鍵 0-9 以及 Enter 鍵。圖 26.18 的第 9 行爲 Keypad 類別成員函式 getInput 的函式原型。因爲這個函式不會改變物件，因此被宣告爲 const。

```
1   // Keypad.h
2   // Keypad class definition. Represents the keypad of the ATM.
3   #ifndef KEYPAD_H
4   #define KEYPAD_H
5
6   class Keypad
7   {
8   public:
9      int getInput() const; // return an integer value entered by user
10  }; // end class Keypad
11
12  #endif // KEYPAD_H
```

圖 26.18　Keypad 類別定義

Keypad 類別成員函式定義

在 Keypad 的實作檔 (圖 26.19) 中，成員函式 getInput (定義在第 9-14 行) 使用標準輸入串流 cin 和串流擷取運算子 >> 取得使用者輸入的資料。第 11 行宣告一個區域變數來儲存使用者的輸入。第 12 行會讀取輸入的資料，存到區域變數 input，接著第 13 行會回傳這個值。getInput 負責取得 ATM 所需的全部輸入。Keypad 的 getInput 成員函式僅簡單地回傳使用者輸入的整數值。假如 Keypad 類別的用戶要求輸入必須滿足某些特殊的條件 (例如，某個數值必須對應到合法的選單選項)，則用戶必須執行適當的錯誤檢查。[請注意：使用標準輸入串流 cin 和字串擷取運算子 >> 會讀取到使用者輸入的非整數資料。因爲眞正的 ATM 小鍵盤只允許整數輸入，因此，我們假設使用者輸入的是一個整數，不處理輸入非整數時所導致的問題。]

```
1  // Keypad.cpp
2  // Member-function definition for class Keypad (the ATM's keypad).
3  #include <iostream>
4  using namespace std;
5
6  #include "Keypad.h" // Keypad class definition
7
8  // return an integer value entered by user
9  int Keypad::getInput() const
10 {
11    int input; // variable to store the input
12    cin >> input; // we assume that user enters an integer
13    return input; // return the value entered by user
14 } // end function getInput
```

圖 26.19　Keypad 類別成員函式定義

26.4.4　CashDispenser 類別

CashDispenser 類別 (圖 26.20-26.21) 代表 ATM 的吐鈔機。圖 26.20 包含預設建構子 (第 9 行) 的函式原型。CashDispenser 類別宣告另外兩個 public 成員函式 dispenseCash (第 12 行) 和 isSufficientCashAvailable (第 15 行)。此類別藉由呼叫 isSufficientCashAvailable 來確認在具有足夠現金時，用戶 (例如 Withdrawal) 才會呼叫 dispenseCash。因此，dispenseCash 只是簡單地模擬發出所需金額的動作，不會檢查是否有足夠的現金。第 17 行宣告 private constant INITIAL_COUNT，代表一開始時在吐鈔機中的鈔票張數 (也就是 500)。第 18 行實作 count 屬性 (模擬在圖 26.11 中)，紀錄 CashDispenser 中剩下的鈔票張數。

```
1  // CashDispenser.h
2  // CashDispenser class definition. Represents the ATM's cash dispenser.
3  #ifndef CASH_DISPENSER_H
4  #define CASH_DISPENSER_H
5
6  class CashDispenser
7  {
8  public:
9     CashDispenser(); // constructor initializes bill count to 500
10
11    // simulates dispensing of specified amount of cash
12    void dispenseCash( int );
13
14    // indicates whether cash dispenser can dispense desired amount
15    bool isSufficientCashAvailable( int ) const;
16 private:
17    static const int INITIAL_COUNT = 500;
18    int count; // number of $20 bills remaining
19 }; // end class CashDispenser
20
21 #endif // CASH_DISPENSER_H
```

圖 26.20　CashDispenser 類別定義

CashDispenser 類別成員函式定義

圖 26.21 包含類別 CashDispenser 成員函式的定義。第 6-9 行的建構子會將 count 設定成初始值 (500)。成員函式 dispenseCash (第 13-17 行) 會模擬吐鈔的行爲。假如我們的系統與眞正的硬體吐鈔機連結，則此成員函式會與實體吐鈔機的硬體裝置互動。這個模擬版本的成員函式會簡單地將剩下的鈔票數量減掉發出 amount 金額所需要的鈔票數量 (第 16 行)。第 15 行會計算發出 amount 金額所需要的 $20 鈔票張數。我們的 ATM 只允許使用者選擇 $20 的倍數作爲提款金額，因此我們將 amount 除以 20，就能得到 billsRequired 的數量。同時，用戶類別 (Withdrawal) 也需負責通知使用者現金已經發出，CashDispenser 不會直接與 Screen 互動。

成員函式 isSufficientCashAvailable (第 20-28 行) 具有參數 amount，代表提領的現金金額。假如 CashDispenser 的 count 大於等於 billsRequired (有足夠的鈔票) ，則第 24-27 行會回傳 true 值，否則會回傳 false 值 (鈔票不夠)。例如，假如使用者想要提領 $80 (也就是說，billsRequired 等於 4)，但是只剩下三張鈔票 (count 等於 3)，則成員函式會回傳 false。

```
1  // CashDispenser.cpp
2  // Member-function definitions for class CashDispenser.
3  #include "CashDispenser.h" // CashDispenser class definition
4
5  // CashDispenser default constructor initializes count to default
6  CashDispenser::CashDispenser()
7  {
8     count = INITIAL_COUNT; // set count attribute to default
9  } // end CashDispenser default constructor
10
11 // simulates dispensing of specified amount of cash; assumes enough cash
12 // is available (previous call to isSufficientCashAvailable returned true)
13 void CashDispenser::dispenseCash( int amount )
14 {
15    int billsRequired = amount / 20; // number of $20 bills required
16    count -= billsRequired; // update the count of bills
17 } // end function dispenseCash
18
19 // indicates whether cash dispenser can dispense desired amount
20 bool CashDispenser::isSufficientCashAvailable( int amount ) const
21 {
22    int billsRequired = amount / 20; // number of $20 bills required
23
24    if ( count >= billsRequired )
25       return true; // enough bills are available
26    else
27       return false; // not enough bills are available
28 } // end function isSufficientCashAvailable
```

圖 26.21　`CashDispenser` 類別成員函式定義

26.4.5　`DepositSlot` 類別

`DepositSlot` 類別 (圖 26.22–26.23) 代表 ATM 的存款槽。跟 `CashDispenser` 類別一樣，這裡的 `DepositSlot` 類別也只是模擬眞實存款槽的功能而已。`DepositSlot` 沒有資料成員，只有一個成員函式 — `isEnvelopeReceived` (宣告在圖 26.22 的第 9 行，定義在圖 26.23 的第 7-10 行)，用來指出是否收到了存款信封。

需求規格書中曾提到，ATM 讓使用者最多有兩分鐘的時間放入信封。因爲這只是一個軟體的模擬，所以目前的 `isEnvelopeReceived` 成員函式版本會立即回傳 `true` 值 (圖 26.23 第 9 行)，我們假設使用者已經在要求的時間內放入信封了。假如有一個眞正的硬體存款槽連接到我們的系統，則 `isEnvelopeReceived` 成員函式會被實作成等待最多兩分鐘的時間，以從存款槽硬體接收到使用者已經插入存款信封的信號。假如 `isEnvelopeReceived` 在兩分鐘內接收到這個信號，則成員函式會回傳 `true` 值。假如經過兩分鐘之後，成員函式仍然沒有收到信號，則會回傳 `false` 值。

```
1  // DepositSlot.h
2  // DepositSlot class definition. Represents the ATM's deposit slot.
3  #ifndef DEPOSIT_SLOT_H
4  #define DEPOSIT_SLOT_H
5
6  class DepositSlot
7  {
8  public:
9     bool isEnvelopeReceived() const; // tells whether envelope was received
10 }; // end class DepositSlot
11
12 #endif // DEPOSIT_SLOT_H
```

圖 26.22　DepositSlot 類別定義

```
1  // DepositSlot.cpp
2  // Member-function definition for class DepositSlot.
3  #include "DepositSlot.h" // DepositSlot class definiton
4
5  // indicates whether envelope was received (always returns true,
6  // because this is only a software simulation of a real deposit slot)
7  bool DepositSlot::isEnvelopeReceived() const
8  {
9     return true; // deposit envelope was received
10 } // end function isEnvelopeReceived
```

圖 26.23　DepositSlot 類別成員函式定義

26.4.6　Account 類別

Account 類別 (圖 26.24-26.25) 代表一個銀行帳戶。類別定義 (圖 26.24) 的第 9-15 行包含此類別的建構子以及六個成員函式的函式原型，我們稍後會討論它們。每個 Account 有四個屬性 (參考圖 26.11 的模型) — accountNumber、pin、availableBalance 和 totalBalance。第 17-20 行將這些屬性實作爲 private 資料成員。資料成員 availableBalance 代表可提領的金額。資料成員 totalBalance 代表可提領的金額，加上存款而等待確認中的金額。

```
1  // Account.h
2  // Account class definition. Represents a bank account.
3  #ifndef ACCOUNT_H
4  #define ACCOUNT_H
5
6  class Account
7  {
8  public:
```

圖 26.24　Account 類別定義

```
 9     Account( int, int, double, double ); // constructor sets attributes
10     bool validatePIN( int ) const; // is user-specified PIN correct?
11     double getAvailableBalance() const; // returns available balance
12     double getTotalBalance() const; // returns total balance
13     void credit( double ); // adds an amount to the Account balance
14     void debit( double ); // subtracts an amount from the Account balance
15     int getAccountNumber() const; // returns account number
16  private:
17     int accountNumber; // account number
18     int pin; // PIN for authentication
19     double availableBalance; // funds available for withdrawal
20     double totalBalance; // funds available + funds waiting to clear
21  }; // end class Account
22
23  #endif // ACCOUNT_H
```

圖 26.24　Account 類別定義 (續)

Account 類別成員函式的定義

圖 26.25 爲 Account 類別成員函式的定義。Account 建構子 (第 6-14 行) 有四個參數，帳號、設定給帳戶的 PIN 碼、可用餘額初值以及總餘額初值。第 8-11 行使用成員初始器將這些值設定給類別的資料成員。

```
 1  // Account.cpp
 2  // Member-function definitions for class Account.
 3  #include "Account.h" // Account class definition
 4
 5  // Account constructor initializes attributes
 6  Account::Account( int theAccountNumber, int thePIN,
 7     double theAvailableBalance, double theTotalBalance )
 8     : accountNumber( theAccountNumber ),
 9       pin( thePIN ),
10       availableBalance( theAvailableBalance ),
11       totalBalance( theTotalBalance )
12  {
13     // empty body
14  } // end Account constructor
15
16  // determines whether a user-specified PIN matches PIN in Account
17  bool Account::validatePIN( int userPIN ) const
18  {
19     if ( userPIN == pin )
20        return true;
21     else
22        return false;
23  } // end function validatePIN
24
25  // returns available balance
```

圖 26.25　Account 類別成員函式定義

```
26  double Account::getAvailableBalance() const
27  {
28     return availableBalance;
29  } // end function getAvailableBalance
30
31  // returns the total balance
32  double Account::getTotalBalance() const
33  {
34     return totalBalance;
35  } // end function getTotalBalance
36
37  // credits an amount to the account
38  void Account::credit( double amount )
39  {
40     totalBalance += amount; // add to total balance
41  } // end function credit
42
43  // debits an amount from the account
44  void Account::debit( double amount )
45  {
46     availableBalance -= amount; // subtract from available balance
47     totalBalance -= amount; // subtract from total balance
48  } // end function debit
49
50  // returns account number
51  int Account::getAccountNumber() const
52  {
53     return accountNumber;
54  } // end function getAccountNumber
```

圖 26.25 Account 類別成員函式定義 (續)

成員函式 validatePIN (第 17-23 行) 用來判斷使用者指定的 PIN 碼 (也就是參數 userPIN) 是否與帳戶的 PIN 碼 (也就是資料成員 pin) 吻合。我們曾在圖 25.19 的 UML 類別示意圖中繪出這個成員函式的參數 userPIN。假如兩個 PIN 碼吻合，則成員函式會回傳 true 值 (第 20 行)，否則會回傳 false 值 (第 22 行)。

成員函式 getAvailableBalance (第 26-29 行) 和 getTotalBalance (第 32-35 行) 爲 get 函式，分別會回傳 double 資料成員 availableBalance 和 totalBalance 的值。

進行存款交易時，成員函式 credit (第 38-41 行) 會將金額 (也就是參數 amount) 增加到帳戶中。請注意，這個成員函式只會將 amount 加到資料成員 totalBalance 中 (第 40 行)。存到帳戶中的金額不會馬上變成可領取的，因此我們只需更改總餘額就好了。我們假設銀行會在稍後自行更新可用餘額。我們的 Account 類別實作只包括能夠執行 ATM 交易的成員函式。因此，我們省略了將資料成員 availableBalance 加

上金額的成員函式 (確認存款) 或是從資料成員 totalBalance 減去金額的成員函式 (拒絕存款) ，雖然在其他的銀行系統中可能會呼叫這些函式。

當執行提款交易時，成員函式 debit (第 44-48 行) 會從 Account 減去某個金額 (也就是參數 amount)。這個成員函式會從資料成員 availableBalance (第 46 行) 和 totalBalance (第 47 行) 中減去金額，因爲提款的動作會同時影響到這兩個帳戶餘額。

成員函式 getAccountNumber (第 51-54 行) 提供存取 Account 的 accountNumber 的服務。我們在實作中加入這個成員函式，讓 BankDatabase 類別的用戶可以辨識出特定的 Account。例如，BankDatabase 擁有許多 Account 物件，它可以在每個 Account 物件上呼叫這個成員函式，以找出具有某個帳號的 Account。

26.4.7　BankDatabase 類別

BankDatabase 類別 (圖 26.26–26.27) 模擬銀行的資料庫，ATM 系統可以與之互動，存取和修改使用者的帳戶資訊。圖 26.26 的類別定義中宣告了類別建構子和幾個成員函式的原型，我們馬上會討論到它們。類別定義中也宣告了 BankDatabase 的資料成員。我們依照 BankDatabase 與 Account 之間的組合關係，決定其中一個資料成員。請回想，圖 26.10 中顯示 BankDatabase 是由零個以上的 Account 類別所組成的。圖 26.26 的第 24 行建立資料成員 accounts (儲存 Account 物件的 vector)，以實作這個組合關係。第 6-7 行允許我們在檔案中使用 vector。第 27 行包含了 private 工具函式 getAccount 的函式原型，讓類別的成員函式可以取得 accounts vector 中的 Account 指標。

```
1  // BankDatabase.h
2  // BankDatabase class definition. Represents the bank's database.
3  #ifndef BANK_DATABASE_H
4  #define BANK_DATABASE_H
5
6  #include <vector> // class uses vector to store Account objects
7  using namespace std;
8
9  #include "Account.h" // Account class definition
10
11 class BankDatabase
12 {
13 public:
14    BankDatabase(); // constructor initializes accounts
15
16    // determine whether account number and PIN match those of an Account
```

圖 26.26　BankDatabase 類別定義

```
17      bool authenticateUser( int, int ); // returns true if Account authentic
18
19      double getAvailableBalance( int ); // get an available balance
20      double getTotalBalance( int ); // get an Account's total balance
21      void credit( int, double ); // add amount to Account balance
22      void debit( int, double ); // subtract amount from Account balance
23   private:
24      vector< Account > accounts; // vector of the bank's Accounts
25
26      // private utility function
27      Account * getAccount( int ); // get pointer to Account object
28   }; // end class BankDatabase
29
30   #endif // BANK_DATABASE_H
```

圖 26.26　BankDatabase 類別定義 (續)

BankDatabase 類別成員函式定義

圖 26.27 包含 BankDatabase 類別的成員函式定義。我們實作一個預設建構子 (第 6-15 行)，將 Account 物件加入資料成員 accounts。爲了測試系統，我們建立兩個內含測試資料的新 Account 物件 (第 9-10 行)，然後將它們加入 vector 的末端 (第 13-14 行)。Account 建構子有四個參數，帳號、設定給帳戶的 PIN 碼、可用餘額初值以及總餘額初值。

```
1   // BankDatabase.cpp
2   // Member-function definitions for class BankDatabase.
3   #include "BankDatabase.h" // BankDatabase class definition
4
5   // BankDatabase default constructor initializes accounts
6   BankDatabase::BankDatabase()
7   {
8      // create two Account objects for testing
9      Account account1( 12345, 54321, 1000.0, 1200.0 );
10     Account account2( 98765, 56789, 200.0, 200.0 );
11
12     // add the Account objects to the vector accounts
13     accounts.push_back( account1 ); // add account1 to end of vector
14     accounts.push_back( account2 ); // add account2 to end of vector
15  } // end BankDatabase default constructor
16
17  // retrieve Account object containing specified account number
18  Account * BankDatabase::getAccount( int accountNumber )
19  {
20     // loop through accounts searching for matching account number
21     for ( size_t i = 0; i < accounts.size(); i++ )
22     {
```

圖 26.27　BankDatabase 類別成員函式定義

```
23          // return current account if match found
24          if ( accounts[ i ].getAccountNumber() == accountNumber )
25             return &accounts[ i ];
26       } // end for
27
28       return NULL; // if no matching account was found, return NULL
29    } // end function getAccount
30
31    // determine whether user-specified account number and PIN match
32    // those of an account in the database
33    bool BankDatabase::authenticateUser( int userAccountNumber,
34       int userPIN )
35    {
36       // attempt to retrieve the account with the account number
37       Account * const userAccountPtr = getAccount( userAccountNumber );
38
39       // if account exists, return result of Account function validatePIN
40       if ( userAccountPtr != NULL )
41          return userAccountPtr->validatePIN( userPIN );
42       else
43          return false; // account number not found, so return false
44    } // end function authenticateUser
45
46    // return available balance of Account with specified account number
47    double BankDatabase::getAvailableBalance( int userAccountNumber )
48    {
49       Account * const userAccountPtr = getAccount( userAccountNumber );
50       return userAccountPtr->getAvailableBalance();
51    } // end function getAvailableBalance
52
53    // return total balance of Account with specified account number
54    double BankDatabase::getTotalBalance( int userAccountNumber )
55    {
56       Account * const userAccountPtr = getAccount( userAccountNumber );
57       return userAccountPtr->getTotalBalance();
58    } // end function getTotalBalance
59
60    // credit an amount to Account with specified account number
61    void BankDatabase::credit( int userAccountNumber, double amount )
62    {
63       Account * const userAccountPtr = getAccount( userAccountNumber );
64       userAccountPtr->credit( amount );
65    } // end function credit
66
67    // debit an amount from Account with specified account number
68    void BankDatabase::debit( int userAccountNumber, double amount )
69    {
70       Account * const userAccountPtr = getAccount( userAccountNumber );
71       userAccountPtr->debit( amount );
72    } // end function debit
```

圖 26.27　BankDatabase 類別成員函式定義 (續)

請回想，BankDatabase 類別是 ATM 類別和眞正 Account 物件的媒介，內含使用者的帳戶資訊。因此，BankDatabase 類別的成員函式只需呼叫屬於目前 ATM 使用者的 Account 物件之對應成員函式。

第 18-29 行是 private 工具函式 getAccount，它讓 BankDatabase 可以取得指向 vector accounts 中某個 Account 的指標。爲了要找到使用者的 Account，BankDatabase 會將成員函式 getAccountNumber 回傳的值和特定的帳號互相比較，直到找到符合的帳戶。第 21-26 行走訪 accounts vector。假如目前 Account (也就是 accounts[i]) 的帳號等於參數 accountNumber 的值，則成員函式會立刻回傳目前 Account 的位址 (也就是指向目前 Account 的指標) 。假如沒有一個帳戶的帳號符合，則第 28 行會回傳 NULL。注意，這個成員函式必須回傳一個指標，而非參照，這是因爲此函式有可能回傳 NULL — 參照不能是 NULL，但是指標可以。

注意，vector 的函式 size (在第 21 行的迴圈測試條件式中呼叫) 會以 size_t 型別 (通常是 unsigned int) 回傳 vector 中的元素數量。因此我們也可以將控制變數 i 宣告爲型別 size_t。若把 i 宣告爲 int，許多編譯器會發出警告訊息，因爲如此一來迴圈測試條件會把有號值 (即 int i) 和無號值 (型別爲 size_t 的值) 進行比較。

成員函式 authenticateUser (第 33-34 行) 會驗證 ATM 使用者的身分。這個函式會接收使用者輸入的帳號和 PIN 碼作爲引數，指出它們是否符合資料庫中的帳號和 PIN 碼。第 37 行呼叫工具函式 getAccount，它會回傳一個指標，指向帳號等於 userAccountNumber 的 Account，或是回傳一個 NULL 表示 userAccountNumber 不合法。我們將 userAccountPtr 宣告爲 const 指標，這是因爲一旦成員函式將指標指向使用者的 Account 之後，指標就不應該改變了。假如 getAccount 回傳一個指向 Account 物件的指標，第 41 行會回傳該物件的 validatePIN 成員函式所回傳的 bool 值。請注意，BankDatabase 的 authenticateUser 成員函式不會執行 PIN 碼的比對，它會將 userPIN 轉給 Account 物件的 validatePIN 成員函式來執行這個工作。Account 成員函式 validatePIN 回傳的值是用來表示使用者輸入的 PIN 碼是否符合 Account 中的 PIN 碼，因此成員函式 authenticateUser 會直接將這個值回傳給類別的客戶 (也就是 ATM)。

在允許使用者執行交易之前，BankDatabase 相信 ATM 已經呼叫成員函式 authenticateUser 並接收回傳的 true 值。BankDatabase 也相信 ATM 建立的每個 Transaction 物件包含的帳號都屬於目前通過驗證的使用者，這個帳號會以 userAccountNumber 引數傳遞給其他的 BankDatabase 成員函式。成員函式 getAvailableBalance (第 47-51 行)、getTotalBalance (第 54-58 行) 以及

`credit` (第 61-65 行) 和 `debit` (第 68-72 行) 只要使用工具函式 `getAccount`，接收指向使用者 `Account` 物件的指標，然後再使用這個指標，在使用者的 `Account` 物件上呼叫適當的 `Account` 成員函式。因爲 `userAccountNumber` 一定會指向存在的 `Account`，因此我們可以確定，這些成員函式中的 `getAccount` 不會回傳 NULL。注意，`getAvailableBalance` 和 `getTotalBalance` 回傳的是對應的 `Account` 成員函式所回傳的值。同時 `credit` 和 `debit` 只是將參數 `amount` 重導到它們所呼叫的 `Account` 成員函式。

26.4.8　`Transaction` 類別

`Transaction` 類別 (圖 26.28–26.29) 是一個抽象基本類別，代表 ATM 的交易。它包含了衍生類別 `BalanceInquiry`、`Withdrawal` 和 `Deposit` 的共同功能。圖 26.28 擴充了 26.3 節建立的 `Transaction` 標頭檔。第 13、17-19 和 22 行包含此類別的建構子以及四個成員函式的函式原型，我們稍後會討論它們。第 15 行定義一個主體爲空的 `virtual` 解構子，這會讓所有衍生類別的解構子都成爲 `virtual` (即使是編譯器自動定義的)，如此一來，當程式經由基本類別的指標刪除動態配置的衍生類別物件時，就可以確保它們都會被妥善地清除。第 24-26 行宣告了此類別的 `private` 資料成員。還記得圖 26.11 的類別示意圖中，`Transaction` 類別包含了 `accountNumber` 屬性 (實作在第 24 行)，代表與此 `Transaction` 有關的帳戶。我們從圖 26.10 中所繪的 `Transaction` 聯繫衍生出資料成員 `screen` (第 25 行) 和 `bankDatabase` (第 26 行)，所有的交易都需要存取 ATM 的螢幕和銀行資料庫，因此我們加入 `Screen` 和 BankDatabase 的參照作爲 `Transaction` 類別的資料成員。你很快會看到 `Transaction` 的建構子初始化這些參照。第 6-7 行的前置宣告表示標頭檔包含類別 `Screen` 和 `BankDatabase` 物件的參照，但是這些類別的定義位在標頭檔之外。

```
1  // Transaction.h
2  // Transaction abstract base class definition.
3  #ifndef TRANSACTION_H
4  #define TRANSACTION_H
5
6  class Screen; // forward declaration of class Screen
7  class BankDatabase; // forward declaration of class BankDatabase
8
9  class Transaction
10 {
11 public:
12    // constructor initializes common features of all Transactions
```

圖 26.28　`Transaction` 類別定義

```
13      Transaction( int, Screen &, BankDatabase & );
14
15      virtual ~Transaction() { } // virtual destructor with empty body
16
17      int getAccountNumber() const; // return account number
18      Screen &getScreen() const; // return reference to screen
19      BankDatabase &getBankDatabase() const; // return reference to database
20
21      // pure virtual function to perform the transaction
22      virtual void execute() = 0; // overridden in derived classes
23   private:
24      int accountNumber; // indicates account involved
25      Screen &screen; // reference to the screen of the ATM
26      BankDatabase &bankDatabase; // reference to the account info database
27   }; // end class Transaction
28
29   #endif // TRANSACTION_H
```

圖 26.28　Transaction 類別定義 (續)

```
1   // Transaction.cpp
2   // Member-function definitions for class Transaction.
3   #include "Transaction.h" // Transaction class definition
4   #include "Screen.h" // Screen class definition
5   #include "BankDatabase.h" // BankDatabase class definition
6
7   // constructor initializes common features of all Transactions
8   Transaction::Transaction( int userAccountNumber, Screen &atmScreen,
9      BankDatabase &atmBankDatabase )
10     : accountNumber( userAccountNumber ),
11       screen( atmScreen ),
12       bankDatabase( atmBankDatabase )
13  {
14     // empty body
15  } // end Transaction constructor
16
17  // return account number
18  int Transaction::getAccountNumber() const
19  {
20     return accountNumber;
21  } // end function getAccountNumber
22
23  // return reference to screen
24  Screen &Transaction::getScreen() const
25  {
26     return screen;
27  } // end function getScreen
28
29  // return reference to bank database
30  BankDatabase &Transaction::getBankDatabase() const
31  {
32     return bankDatabase;
33  } // end function getBankDatabase
```

圖 26.29　Transaction 類別成員函式定義

Transaction 類別有一個建構子 (宣告在圖 26.28 的第 13 行，定義在圖 26.29 的第 8-15 行)，它會接收目前使用者的帳號和 ATM 螢幕和銀行資料庫的參照作爲引數。因爲 Transaction 是一個抽象類別，程式永遠不會直接呼叫這個建構子來實體化 Transaction 物件。Transaction 衍生類別的建構子會使用基本類別初始值語法來呼叫這個建構子。

Transaction 類別有三個 public get 函式 －getAccountNumber (宣告在圖 26.28 的第 17 行，定義在圖 26.29 的第 18-21 行)、getScreen (宣告在圖 26.28 的第 18 行，定義在圖 26.29 的第 24-27 行) 以及 getBankDatabase (宣告在圖 26.28 的第 19 行，定義在圖 26.29 的第 30-33 行)。Transaction 的衍生類別從 Transaction 那裡繼承了這些成員函式，利用它們存取 Transaction 類別的 private 資料成員。

Transaction 類別也宣告了 pure virtual 函式 execute (圖 26.28 第 22 行)。替這個成員函式提供實作是沒有意義的，因爲一般化的交易不能眞的執行。因此，我們將這個成員函式宣告爲 pure virtual 函式，強迫每個 Transaction 的衍生類別提供它自己的具象實作，以執行特定類型的交易。

26.4.9　BalanceInquiry 類別

BalanceInquiry 類別 (圖 26.30–26.31) 衍生自抽象基本類別 Transaction，代表 ATM 的查詢餘額交易。BalanceInquiry 沒有它自己的資料成員，但是它繼承了 Transaction 的資料成員 accountNumber、screen 和 bankDatabase，可以透過 Transaction 的 public get 函式存取這些資料。第 6 行#include 了 Transaction 基本類別的定義。BalanceInquiry 建構子 (宣告在圖 26.30 的第 11 行，定義在圖 26.31 的第 8-13 行) 會接收對應到 Transaction 資料成員的引數，然後利用基本類別初始值語法，將它們轉送到 Transaction 的建構子 (圖 26.31 第 10 行)。圖 26.30 第 12 行包含成員函式 execute 的函式原型，用來指出程式想要重載同名的基本類別 pure virtual 函式。

BalanceInquiry 類別重載 Transaction 的 pure virtual 函式 execute，提供具象實作 (圖 26.31 的第 16-37 行) ，執行查詢餘額的步驟。第 19–20 行呼叫繼承自 Transaction 基本類別的成員函式，取得銀行資料庫和 ATM 螢幕的參照。第 23-24 行呼叫 bankDatabase 的成員函式 getAvailableBalance，取得帳戶的餘額。第 24 行使用繼承的成員函式 getAccountNumber，取得目前使用者的帳號，然後將它傳給 getAvailableBalance。第 27-28 行取得目前使用者帳戶的總餘額。第 31-36 行會在

ATM 螢幕上顯示餘額的資訊。還記得 `displayDollarAmount` 會接收一個 `double` 引數，以金錢格式將它輸出到螢幕上。例如，假如使用者的 `availableBalance` 是 700.5，則第 33 行會輸出 $700.50。第 36 行會插入一行空白，將餘額資訊與接下來的輸出分開(在執行 `BalanceInquiry` 之後，ATM 類別會再次輸出主選單)。

```
1  // BalanceInquiry.h
2  // BalanceInquiry class definition. Represents a balance inquiry.
3  #ifndef BALANCE_INQUIRY_H
4  #define BALANCE_INQUIRY_H
5
6  #include "Transaction.h" // Transaction class definition
7
8  class BalanceInquiry : public Transaction
9  {
10 public:
11    BalanceInquiry( int, Screen &, BankDatabase & ); // constructor
12    virtual void execute(); // perform the transaction
13 }; // end class BalanceInquiry
14
15 #endif // BALANCE_INQUIRY_H
```

圖 26.30　`BalanceInquiry` 類別定義

```
1  // BalanceInquiry.cpp
2  // Member-function definitions for class BalanceInquiry.
3  #include "BalanceInquiry.h" // BalanceInquiry class definition
4  #include "Screen.h" // Screen class definition
5  #include "BankDatabase.h" // BankDatabase class definition
6
7  // BalanceInquiry constructor initializes base-class data members
8  BalanceInquiry:: BalanceInquiry( int userAccountNumber, Screen &atmScreen,
9     BankDatabase &atmBankDatabase )
10    : Transaction( userAccountNumber, atmScreen, atmBankDatabase )
11 {
12    // empty body
13 } // end BalanceInquiry constructor
14
15 // performs transaction; overrides Transaction's pure virtual function
16 void BalanceInquiry::execute()
17 {
18    // get references to bank database and screen
19    BankDatabase &bankDatabase = getBankDatabase();
20    Screen &screen = getScreen();
21
22    // get the available balance for the current user's Account
23    double availableBalance =
24       bankDatabase.getAvailableBalance( getAccountNumber() );
25
```

圖 26.31　`BalanceInquiry` 類別成員函式定義

```
26      // get the total balance for the current user's Account
27      double totalBalance =
28         bankDatabase.getTotalBalance( getAccountNumber() );
29
30      // display the balance information on the screen
31      screen.displayMessageLine( "\nBalance Information:" );
32      screen.displayMessage( " - Available balance: " );
33      screen.displayDollarAmount( availableBalance );
34      screen.displayMessage( "\n - Total balance:     " );
35      screen.displayDollarAmount( totalBalance );
36      screen.displayMessageLine( "" );
37   } // end function execute
```

圖 26.31　`BalanceInquiry` 類別成員函式定義 (續)

26.4.10　`Withdrawal` 類別

`Withdrawal` 類別 (圖 26.32–26.33) 衍生自 `Transaction`，代表 ATM 的提款交易。圖 26.32 擴充圖 26.13 建立的類別標頭檔。`Withdrawal` 有一個建構子及一個成員函式 `execute`，我們稍後會討論它們。從圖 26.11 的類別示意圖可以得知，`Withdrawal` 類別有一個屬性 `amount`，第 16 行會將它時作成一個 `int` 資料成員。圖 26.10 也繪出 `Withdrawal` 類別和 `Keypad` 與 `CashDispenser` 類別之間的聯繫，因此第 17-18 行分別實作了 `keypad` 和 `cashDispenser` 的參照。第 19 行是一個 `private` 工具函式的函式原型，我們很快會討論到它。

```
1   // Withdrawal.h
2   // Withdrawal class definition. Represents a withdrawal transaction.
3   #ifndef WITHDRAWAL_H
4   #define WITHDRAWAL_H
5
6   #include "Transaction.h" // Transaction class definition
7   class Keypad; // forward declaration of class Keypad
8   class CashDispenser; // forward declaration of class CashDispenser
9
10  class Withdrawal : public Transaction
11  {
12  public:
13     Withdrawal( int, Screen &, BankDatabase &, Keypad &, CashDispenser & );
14     virtual void execute(); // perform the transaction
15  private:
16     int amount; // amount to withdraw
17     Keypad &keypad; // reference to ATM's keypad
18     CashDispenser &cashDispenser; // reference to ATM's cash dispenser
19     int displayMenuOfAmounts() const; // display the withdrawal menu
20  }; // end class Withdrawal
21
22  #endif // WITHDRAWAL_H
```

圖 26.32　`Withdrawal` 類別定義

`Withdrawal` 類別成員函式定義

圖 26.33 包含 `Withdrawal` 類別的成員函式定義。第 3 行 `#include` 類別定義，第 4-7 行 `#include Withdrawal` 成員函式會用到的其他類別的定義。第 11 行宣告一個全域常數，對應到提款選單的取消選項。我們很快會見到類別如何利用這個常數。

```
1  // Withdrawal.cpp
2  // Member-function definitions for class Withdrawal.
3  #include "Withdrawal.h" // Withdrawal class definition
4  #include "Screen.h" // Screen class definition
5  #include "BankDatabase.h" // BankDatabase class definition
6  #include "Keypad.h" // Keypad class definition
7  #include "CashDispenser.h" // CashDispenser class definition
8
9  // global constant that corresponds to menu option to cancel
10 static const int CANCELED = 6;
11
12 // Withdrawal constructor initialize class's data members
13 Withdrawal::Withdrawal( int userAccountNumber, Screen &atmScreen,
14    BankDatabase &atmBankDatabase, Keypad &atmKeypad,
15    CashDispenser &atmCashDispenser )
16    : Transaction( userAccountNumber, atmScreen, atmBankDatabase ),
17      keypad( atmKeypad ), cashDispenser( atmCashDispenser )
18 {
19    // empty body
20 } // end Withdrawal constructor
21
22 // perform transaction; overrides Transaction's pure virtual function
23 void Withdrawal::execute()
24 {
25    bool cashDispensed = false; // cash was not dispensed yet
26    bool transactionCanceled = false; // transaction was not canceled yet
27
28    // get references to bank database and screen
29    BankDatabase &bankDatabase = getBankDatabase();
30    Screen &screen = getScreen();
31
32    // loop until cash is dispensed or the user cancels
33    do
34    {
35       // obtain the chosen withdrawal amount from the user
36       int selection = displayMenuOfAmounts();
37
38       // check whether user chose a withdrawal amount or canceled
39       if ( selection != CANCELED )
40       {
41          amount = selection; // set amount to the selected dollar amount
42
43          // get available balance of account involved
```

圖 26.33　`Withdrawal` 類別成員函式定義

```
44            double availableBalance =
45               bankDatabase.getAvailableBalance( getAccountNumber() );
46
47            // check whether the user has enough money in the account
48            if ( amount <= availableBalance )
49            {
50               // check whether the cash dispenser has enough money
51               if ( cashDispenser.isSufficientCashAvailable( amount ) )
52               {
53                  // update the account involved to reflect withdrawal
54                  bankDatabase.debit( getAccountNumber(), amount );
55
56                  cashDispenser.dispenseCash( amount ); // dispense cash
57                  cashDispensed = true; // cash was dispensed
58
59                  // instruct user to take cash
60                  screen.displayMessageLine(
61                     "\nPlease take your cash from the cash dispenser." );
62               } // end if
63               else // cash dispenser does not have enough cash
64                  screen.displayMessageLine(
65                     "\nInsufficient cash available in the ATM."
66                     "\n\nPlease choose a smaller amount." );
67            } // end if
68            else // not enough money available in user's account
69            {
70               screen.displayMessageLine(
71                  "\nInsufficient funds in your account."
72                  "\n\nPlease choose a smaller amount." );
73            } // end else
74         } // end if
75         else // user chose cancel menu option
76         {
77            screen.displayMessageLine( "\nCanceling transaction..." );
78            transactionCanceled = true; // user canceled the transaction
79         } // end else
80      } while ( !cashDispensed && !transactionCanceled ); // end do...while
81   } // end function execute
82
83   // display a menu of withdrawal amounts and the option to cancel;
84   // return the chosen amount or 0 if the user chooses to cancel
85   int Withdrawal::displayMenuOfAmounts() const
86   {
87      int userChoice = 0; // local variable to store return value
88
89      Screen &screen = getScreen(); // get screen reference
90
91      // array of amounts to correspond to menu numbers
92      int amounts[] = { 0, 20, 40, 60, 100, 200 };
93
94      // loop while no valid choice has been made
95      while ( userChoice == 0 )
96      {
```

圖 26.33　`Withdrawal` 類別成員函式定義 (續 1)

```
97          // display the menu
98          screen.displayMessageLine( "\nWithdrawal options:" );
99          screen.displayMessageLine( "1 - $20" );
100         screen.displayMessageLine( "2 - $40" );
101         screen.displayMessageLine( "3 - $60" );
102         screen.displayMessageLine( "4 - $100" );
103         screen.displayMessageLine( "5 - $200" );
104         screen.displayMessageLine( "6 - Cancel transaction" );
105         screen.displayMessage( "\nChoose a withdrawal option (1-6): " );
106
107         int input = keypad.getInput(); // get user input through keypad
108
109         // determine how to proceed based on the input value
110         switch ( input )
111         {
112            case 1: // if the user chose a withdrawal amount
113            case 2: // (i.e., chose option 1, 2, 3, 4 or 5), return the
114            case 3: // corresponding amount from amounts array
115            case 4:
116            case 5:
117               userChoice = amounts[ input ]; // save user's choice
118               break;
119            case CANCELED: // the user chose to cancel
120               userChoice = CANCELED; // save user's choice
121               break;
122            default: // the user did not enter a value from 1-6
123               screen.displayMessageLine(
124                  "\nInvalid selection. Try again." );
125         } // end switch
126      } // end while
127
128      return userChoice; // return withdrawal amount or CANCELED
129   } // end function displayMenuOfAmounts
```

圖 26.33　`Withdrawal` 類別成員函式定義 (續 2)

`Withdrawal` 類別的建構子 (定義在圖 26.33 的第 13-20 行) 有五個參數。第 16 行使用基本類別初始值，將參數 `userAccountNumber`、`atmScreen` 和 `atmBankDatabase` 傳給基本類別 `Transaction` 的建構子，以設定繼承自 `Transaction` 的資料成員。建構子也接收 `atmKeypad` 和 `atmCashDispenser` 參照作爲參數，並利用成員初始器將它們設定給資料成員 `keypad` 和 `cashDispenser` (第 17 行)。

`Withdrawal` 類別重載 `Transaction` 的 `pure virtual` 函式 `execute`，提供具象實作 (第 23-81 行)，執行提款的步驟。第 25 行宣告了區域 `bool` 變數 `cashDispensed`，並且設定其初始值。這個變數指出是否已經發出現金 (也就是交易是否完全成功)，其初始值爲 `false`。第 26 行會宣告 `bool` 變數 `transactionCanceled`，並將它初始化，這

個變數會指出使用者是否取消交易。第 29-30 行呼叫繼承自 Transaction 基本類別的成員函式，取得銀行資料庫和 ATM 螢幕的參照。

第 33-80 行包含一個 do...while 敘述，會重複執行直到發出現金 (也就是說，直到 cashDispensed 變成 ture 值) 或是直到使用者選擇取消 (也就是說，直到 transactionCanceled 變成 true 值) 爲止。假如發生錯誤 (也就是說，想要提領的金額大於使用者的可用餘額或是吐鈔機中的現金)，迴圈會重複回到使用者，重新開始交易。第 36 行呼叫 private 工具函式 displayMenuOfAmounts (定義在第 85-129 行)，顯示提款金額的選單，取得使用者的選擇。這個函式會顯示金額選單，並回傳 int 提款金額或是 int 常數 CANCELED，表示使用者選擇取消交易。

成員函式 displayMenuOfAmounts (第 85-129 行) 會先宣告區域變數 userChoice (初始值爲 0)，儲存成員函式回傳的值 (第 87 行)。第 89 行呼叫繼承自基本類別 Transaction 的成員函式 getScreen，取得螢幕的參照。第 92 行宣告一個整數陣列，內含對應到提款選單的提款金額。我們忽略陣列的第一個元素 (索引 0)，因爲選單沒有 0 選項。第 95-126 行的 while 敘述會一直重複，直到 userChoice 選擇 0 以外的數值。我們稍後會討論，當使用者選擇合法的選項時，會發生的事。第 98-105 行會在螢幕上顯示提款選單，並提示使用者輸入選項。第 107 行透過鍵盤取得整數 input。第 110-125 行的 switch 敘述決定如何根據使用者的輸入來進行接下來的動作。假如使用者選擇 1-5 之間的數字，第 117 行會將 userChoice 設定爲陣列 amounts 中索引爲 input 的元素的值。例如，假設使用者輸入 3，想要提領$60，則第 117 行會將 userChoice 設定爲 amounts[3]的值 (也就是 60)。第 118 行會結束 switch 敘述。此時 userChoice 不等於 0 了，因此第 95-126 行的 while 迴圈會結束，第 128 行會回傳 userChoice。假如使用者選擇了取消，則第 120-121 行會執行，將 userChoice 設定爲 CANCELED，導致成員函式回傳此值。假如使用者沒有輸入合法的選單選項，則第 123-124 行會顯示一個錯誤訊息，接著使用者會回到提款選單。

在 execute 成員函式中，第 39 行的 if 敘述會判斷使用者輸入了提領金額或是選擇取消。假如使用者選擇取消，則會執行第 77-78 行，向使用者顯示適當的訊息，並將 transactionCanceled 設定爲 true。這樣會讓第 80 行的迴圈測試條件失敗，而控制流程回到呼叫函式 (也就是 ATM 的成員函式 performTransactions)。假如使用者選擇了提款金額，第 41 行會將區域變數 selection 的值設定給資料成員 amount。第 44-45 行取得目前使用者帳戶中的可用餘額，將它存在區域 double 變數 availableBalance 中。接著第 48 行的 if 敘述會判斷提取金額是否小於等於使用者

帳戶的可用餘額。假如不是，則第 70-72 行會顯示一個錯誤訊息。接著控制流程會走到 `do...while` 的尾端，然後重複迴圈 (因爲 `cashDispensed` 和 `transactionCanceled` 仍爲 `false`)。假如使用者的餘額足夠，則第 51 行的 `if` 敘述會呼叫 `cashDispenser` 的 `isSufficientCashAvailable` 成員函式，判斷吐鈔機是否有足夠的金額來滿足提款需求。假如這個成員函式回傳 `false` 值，則第 64-66 行會顯示一個錯誤訊息，接著 `do-while` 會繼續重複執行。假如有足夠的現金可以滿足提領的需求，第 54 行會從使用者帳戶的資料庫中減去提領的金額。第 56-57 行會指示吐鈔機發出現金，並將 `cashDispensed` 設定爲 `true`。最後，第 60-61 行會顯示一個訊息，告訴使用者機器已經吐出現金了。因爲 `cashDispensed` 現在是 `true` 了，因此控制流程會從 `do...while` 敘述之後開始。沒有其他敘述式出現在迴圈之後，因此成員函式會返回到 ATM 類別。

在第 64-66 行和第 70-72 行的函式呼叫中，我們將傳遞給 `Screen` 成員函式 `displayMessageLine` 的引數分爲兩個字串常數，印成兩行。這是因爲這個引數太長，單一行無法容納。C++ 會把相鄰的字串字面接起來，就算它們出現在不同行。例如，假如你寫的是`"Happy" "Birthday"`，C++會將這兩個相鄰的字串常數視爲一個字串常數 `"Happy Birthday"`。因此，當第 64-66 行執行時，`displayMessageLine` 會接收單一 `string` 作爲參數，即使此函式中的引數看起來是兩個字串常數。

26.4.11 `Deposit` 類別

`DepositSlot` 類別 (圖 26.34–26.35) 衍生自 `Transaction`，代表 ATM 的存款交易。圖 26.34 包含 `Deposit` 類別的定義。如同衍生類別 `BalanceInquiry` 和 `Withdrawal`，`Deposit` 宣告一個建構子 (第 13 行) 和成員函式 `execute` (第 14 行)，我們稍後會討論它們。還記得圖 26.11 的類別示意圖中，`Deposit` 類別包含了 `amount` 屬性，第 16 行將它實作爲 `int` 資料成員。第 17-18 行建立了參照資料成員 keypad 和 `depositSlot`，實作類別圖 26.10 中 `Deposit` 與 `Keypad` 和 `DepositSlot` 類別的聯繫。第 19 行是一個 `private` 工具函式 `promptForDepositAmount` 的函式原型，我們很快會討論到它。

```
1  // Deposit.h
2  // Deposit class definition. Represents a deposit transaction.
3  #ifndef DEPOSIT_H
4  #define DEPOSIT_H
5
6  #include "Transaction.h" // Transaction class definition
7  class Keypad; // forward declaration of class Keypad
8  class DepositSlot; // forward declaration of class DepositSlot
9
10 class Deposit : public Transaction
11 {
12 public:
13    Deposit( int, Screen &, BankDatabase &, Keypad &, DepositSlot & );
14    virtual void execute(); // perform the transaction
15 private:
16    double amount; // amount to deposit
17    Keypad &keypad; // reference to ATM's keypad
18    DepositSlot &depositSlot; // reference to ATM's deposit slot
19    double promptForDepositAmount() const; // get deposit amount from user
20 }; // end class Deposit
21
22 #endif // DEPOSIT_H
```

圖 26.34　`Deposit` 類別定義

`Deposit` 類別成員函式的定義

圖 26.35 爲 `Deposit` 類別的實作。第 3 行 `#include Deposit` 類別定義，第 4-7 行 `#include Deposit` 成員函式會用到的其他類別定義。第 9 行宣告常數 `CANCELED`，這個值用來表示使用者想要取消存款交易。我們很快會見到類別如何利用這個常數。

```
1  // Deposit.cpp
2  // Member-function definitions for class Deposit.
3  #include "Deposit.h" // Deposit class definition
4  #include "Screen.h" // Screen class definition
5  #include "BankDatabase.h" // BankDatabase class definition
6  #include "Keypad.h" // Keypad class definition
7  #include "DepositSlot.h" // DepositSlot class definition
8
9  static const int CANCELED = 0; // constant representing cancel option
10
11 // Deposit constructor initializes class's data members
12 Deposit::Deposit( int userAccountNumber, Screen &atmScreen,
13    BankDatabase &atmBankDatabase, Keypad &atmKeypad,
14    DepositSlot &atmDepositSlot )
15    : Transaction( userAccountNumber, atmScreen, atmBankDatabase ),
16      keypad( atmKeypad ), depositSlot( atmDepositSlot )
17 {
```

圖 26.35　`Deposit` 類別成員函式的定義

```
18      // empty body
19   } // end Deposit constructor
20
21   // performs transaction; overrides Transaction's pure virtual function
22   void Deposit::execute()
23   {
24      BankDatabase &bankDatabase = getBankDatabase(); // get reference
25      Screen &screen = getScreen(); // get reference
26
27      amount = promptForDepositAmount(); // get deposit amount from user
28
29      // check whether user entered a deposit amount or canceled
30      if ( amount != CANCELED )
31      {
32         // request deposit envelope containing specified amount
33         screen.displayMessage(
34            "\nPlease insert a deposit envelope containing " );
35         screen.displayDollarAmount( amount );
36         screen.displayMessageLine( " in the deposit slot." );
37
38         // receive deposit envelope
39         bool envelopeReceived = depositSlot.isEnvelopeReceived();
40
41         // check whether deposit envelope was received
42         if ( envelopeReceived )
43         {
44            screen.displayMessageLine( "\nYour envelope has been received."
45               "\nNOTE: The money deposited will not be available until we"
46               "\nverify the amount of any enclosed cash, and any enclosed "
47               "checks clear." );
48
49            // credit account to reflect the deposit
50            bankDatabase.credit( getAccountNumber(), amount );
51         } // end if
52         else // deposit envelope not received
53         {
54            screen.displayMessageLine( "\nYou did not insert an "
55               "envelope, so the ATM has canceled your transaction." );
56         } // end else
57      } // end if
58      else // user canceled instead of entering amount
59      {
60         screen.displayMessageLine( "\nCanceling transaction..." );
61      } // end else
62   } // end function execute
63
64   // prompt user to enter a deposit amount in cents
65   double Deposit::promptForDepositAmount() const
66   {
67      Screen &screen = getScreen(); // get reference to screen
68
69      // display the prompt and receive input
```

圖 26.35 Deposit 類別成員函式的定義 (續 1)

```
70      screen.displayMessage( "\nPlease enter a deposit amount in "
71         "CENTS (or 0 to cancel): " );
72      int input = keypad.getInput(); // receive input of deposit amount
73
74      // check whether the user canceled or entered a valid amount
75      if ( input == CANCELED )
76         return CANCELED;
77      else
78      {
79         return static_cast< double >( input ) / 100; // return dollar amount
80      } // end else
81   } // end function promptForDepositAmount
```

圖 26.35　Deposit 類別成員函式的定義 (續 2)

如同 Withdrawal 類別，Deposit 類別包含一個建構子 (第 12-19 行)，它會利用基本類別初始值傳遞三個參數給基本類別 Transaction 的建構子 (第 15 行)。此建構子還有參數 atmKeypad 和 atmDepositSlot，會設定給對應的資料成員 (第 16 行)。

execute 成員函式 (第 22-62 行) 重載基本類別 Transaction 的 pure virtual 函式 execute，執行存款的步驟。第 24-25 行取得資料庫和螢幕的參照。第 27 行呼叫 private 工具函式 promptForDepositAmount (定義在第 65-81 行)，提示使用者輸入存款金額，並將回傳的金額設定給 amount 資料成員。成員函式 promptForDepositAmount 會要求使用者以分爲單位，輸入整數的存款金額 (因爲 ATM 的小鍵盤不包含小數點，許多眞實的 ATM 都是如此)，並將代表存款金額的 double 值回傳。

成員函式 promptForDepositAmount 的第 67 行會取得 ATM 螢幕的參照。第 70-71 行會在螢幕上顯示一個訊息，要求使用者以分爲單位，輸入存款金額，或是輸入 0 取消交易。第 72 行接收使用者從鍵盤輸入的資料。第 75-80 行的 if 敘述會判斷使用者輸入了提款金額或是選擇取消交易。假如使用者選擇取消，則第 76 行會回傳常數 CANCELED。否則，第 79 行會將 input 轉型爲 double，然後除以 100，將單位從美分換算爲美元，接著回傳存款金額。例如，假如使用者輸入了 125，單位爲分，則第 79 行會回傳 125.0 除以 100，也就是 1.25 (125 分等於 1.25 元) 。

在 execute 成員函式中，第 30-61 行的 if 敘述會判斷使用者是否選擇取消交易而非輸入存款金額。若使用者取消交易，第 60 行會顯示適當訊息，然後返回。假如使用者輸入了存款金額，第 33-36 行會指示使用者放入內含該金額的存款信封。還記得 Screen 成員函式 displayDollarAmount 會輸出一個金錢格式的 double 值。

第 39 行會將區域 bool 變數設定爲 depositSlot 的 isEnvelope 成員函式回傳的值，表示是否收到了存款信封。還記得我們讓 isEnvelopeReceived (圖 26.23 的第

7-10 行) 永遠會回傳 `true` 值，因爲我們是模擬存款槽的功能，假設使用者一定會放入信封。然而，我們還是讓 `Deposit` 的成員函式 `execute` 判斷使用者是否放入信封，良好的軟體工程應該考慮到所有可能的回傳值。因此，`Deposit` 已經準備好要處理未來可能回傳 `false` 值的 `isEnvelopeReceived` 版本。假如存款槽收到信封，則第 44-50 行會執行。第 44-47 行會對使用者顯示適當的訊息。第 50 行會在使用者的帳戶中加入存入的金額。假如存款槽沒有收到信封，則第 54-55 行會執行。在這種情況下，我們會向使用者顯示一個訊息，表示 ATM 已經取消交易了。接著成員函式會返回，不會修改使用者帳戶中的值。

26.4.12　測試程式 `ATMCaseStudy.cpp`

`ATMCaseStudy.cpp` (圖 26.36) 是一個簡單的 C++ 程式，讓使用者能夠啓動 ATM，並測試我們的 ATM 系統模型實作。程式的 `main` 函式 (第 6-11 行) 只是建立了名爲 `atm` (第 8 行) 的新 ATM 物件，接著呼叫它的 `run` 成員函式 (第 9 行) 以啓動 ATM 系統。

```
1  // ATMCaseStudy.cpp
2  // Driver program for the ATM case study.
3  #include "ATM.h" // ATM class definition
4
5  // main function creates and runs the ATM
6  int main()
7  {
8     ATM atm; // create an ATM object
9     atm.run(); // tell the ATM to start
10 } // end main
```

圖 26.36　`ATMCaseStudy.cpp` 啓動 ATM 系統

26.5　總結

在本章中，我們使用繼承進一步改良 ATM 軟體系統的設計，然後完整地實作 ATM 的 C++ 程式。恭喜你完成了整個 ATM 的案例研討！我們希望這個學習經驗能幫助你，加強物件導向程式設計的概念。在下一章中，我們將介紹如何利用 OGRE 設計遊戲程式。

自我測驗習題解答

26.1 對。減號代表可見度爲 private。類別間的「夥伴關係」(friendship) 會打破這個限制；我們將在第 10 章討論這個主題。

26.2 b.

26.3 設計 Account 類別所產生的標頭檔如圖 26.37。

```
1  // Fig. 26.37: Account.h
2  // Account class definition. Represents a bank account.
3  #ifndef ACCOUNT_H
4  #define ACCOUNT_H
5
6  class Account
7  {
8  public:
9     bool validatePIN( int ); // is user-specified PIN correct?
10    double getAvailableBalance(); // returns available balance
11    double getTotalBalance(); // returns total balance
12    void credit( double ); // adds an amount to the Account
13    void debit( double ); // subtracts an amount from the Account
14 private:
15    int accountNumber; // account number
16    int pin; // PIN for authentication
17    double availableBalance; // funds available for withdrawal
18    double totalBalance; // funds available + funds waiting to clear
19 }; // end class Account
20
21 #endif // ACCOUNT_H
```

圖 26.37　根據圖 26.1 和 26.2 實作的 Account 類別標頭檔

26.4 b.

26.5 錯。UML 規定我們要將所有的抽象類別跟抽象操作名稱以斜體字呈現。

26.6 設計 Transaction 類別所產生的標頭檔如圖 26.38。在實作中，建構子會初始化實體物件的 private 參照屬性 screen 和 bankDatabase，而成員函式 getScreen 和 getBankDatabase 可以存取這些屬性。這些成員函式允許 Transaction 的衍生類別存取 ATM 的 screen，並可以與銀行的資料庫連線互動。

```
1  // Fig. 36.38: Transaction.h
2  // Transaction abstract base class definition.
3  #ifndef TRANSACTION_H
4  #define TRANSACTION_H
5
6  class Screen; // forward declaration of class Screen
7  class BankDatabase; // forward declaration of class BankDatabase
8
9  class Transaction
10 {
11 public:
```

圖 26.38　根據圖 26.10 和圖 26.11 所設計的 Transaction 類別標頭檔

```
12     int getAccountNumber(); // return account number
13     Screen &getScreen(); // return reference to screen
14     BankDatabase &getBankDatabase(); // return reference to bank database
15
16     // pure virtual function to perform the transaction
17     virtual void execute() = 0; // overridden in derived classes
18  private:
19     int accountNumber; // indicates account involved
20     Screen &screen; // reference to the screen of the ATM
21     BankDatabase &bankDatabase; // reference to the account info database
22  }; // end class Transaction
23
24  #endif // TRANSACTION_H
```

圖 26.38　根據圖 26.10 和圖 26.11 所設計的 `Transaction` 類別標頭檔 (續)