APPENDICE E

Il codice del progetto ATM

E.1 Implementazione del progetto ATM

Questa appendice contiene il codice completo dell'implementazione del progetto del sistema ATM, sviluppato nelle sezioni "Pensare in termini di oggetti" alla fine dei capitoli 1-7, 9 e 13. Il codice C++ dell'implementazione comprende complessivamente 877 righe. Consideriamo le classi nell'ordine in cui sono state identificate nella sezione 3.11:

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

Nell'implementazione delle classi vengono considerate le linee guida introdotte nelle sezioni 9.12 e 13.10 e i modelli UML delle figure 13.28 e 13.29. Per sviluppare la definizione delle funzioni membro si fa riferimento ai diagrammi di attività della sezione 5.11 e ai diagrammi di sequenza e di comunicazione della sezione 7.12. Osservate che il nostro progetto ATM non specifica tutta la logica del programma e può anche non specificare completamente gli attributi e le operazioni richieste per completare l'implementazione. Ciò costituisce una normale fase dello sviluppo di un progetto orientato agli oggetti. Nel corso dell'implementazione del sistema viene completata la logica del programma e vengono aggiunti attributi e comportamenti necessari a soddisfare le specifiche della sezione 2.8.

Completiamo la discussione introducendo un programma C++ (`ATMCaseStudy.cpp`) che permette di testare tutte le classi del sistema. Ricordate che stiamo eseguendo una prima versione dell'applicazione, che gira su un personal computer e simula il tastierino e lo schermo della macchina ATM con la tastiera e lo schermo di un personal computer. Anche il funzionamento degli sportelli per il deposito e il prestito sono solo simulati; l'implementazione, tuttavia, è tale per cui le versioni reali di questi componenti possono essere integrate nel sistema senza sforzo.

E.2 La classe ATM

La classe `ATM` (figure E.1-E.2) rappresenta il sistema nel suo complesso. La figura E.1 contiene la definizione della classe, racchiusa tra le direttive per il preprocessore `#ifndef`, `#define` e `#endif` per evitare inclusioni multiple della classe nel programma. Le righe 16-17 contengono i prototipi delle funzioni membro della classe. Il diagramma delle classi di figura 13.29 non prevede operazioni per la classe `ATM` ma viene inclusa la funzione membro pubblica `run` (riga 17) per permettere ad un agente esterno (`ATMCaseStudy.cpp`) di eseguire l'applicazione. Viene inoltre incluso il prototipo per un costruttore di default (riga 16).

```
1   // ATM.h
2   // Definizione della classe ATM che rappresenta un bancomat.
3   #ifndef ATM_H
4   #define ATM_H
5
6   #include "Screen.h" // definizione della classe Screen
7   #include "Keypad.h" // definizione della classe Keypad
8   #include "CashDispenser.h" // definizione della classe CashDispenser
9   #include "DepositSlot.h" // definizione della classe DepositSlot
10  #include "BankDatabase.h" // definizione della classe BankDatabase
11  class Transaction; // dichiarazione anticipata della classe Transaction
12
13  class ATM
14  {
15  public:
16     ATM(); // costruttore che inizializza i dati membro
17     void run(); // mette in funzione la macchina ATM
18  private:
19     bool userAuthenticated; // utente autenticato?
20     int currentAccountNumber; // numero di conto corrente dell'utente attuale
21     Screen screen; // schermo della macchina ATM
22     Keypad keypad; // tastierino dell'ATM
23     CashDispenser cashDispenser; // sportello dei prelievi dell'ATM
24     DepositSlot depositSlot; // sportello dei depositi dell'ATM
25     BankDatabase bankDatabase; // database della banca
26
27     // funzioni di utilità private
28     void authenticateUser(); // autentica un'utente
29     void performTransactions(); // esegue le transazioni
30     int displayMainMenu() const; // visualizza il menu principale
31
32     // restituisce un oggetto della specifica classe derivata da Transaction
33     Transaction *createTransaction( int );
34  }; // fine della classe ATM
35
36  #endif // ATM_H
```

Figura E.1 Definizione della classe ATM che rappresenta il sistema nel suo complesso

Le righe 19-25 di figura E.1 implementano gli attributi ella classe come membri `private`. Tutti gli attributi (tranne uno) sono stati identificati nei diagrammi delle classi delle figure 13.28 e 13.29. Si noti che l'attributo `userAuthenticated` di tipo UML `Boolean` in figura 13.29 viene implementato in C++ con un dato membro di tipo `bool` (riga 19). La riga 20 dichiara un dato membro non menzionato nei diagrammi UML, l'attributo intero `currentAccountNumber` che tiene traccia del numero di conto corrente dell'utente autenticato. Vedremo in seguito come viene utilizzato questo attributo.

Le righe 21-24 creano gli oggetti che rappresentano le parti del sistema ATM. Notate che nel diagramma delle classi di figura 13.28 viene evidenziato che la classe `ATM` possiede relazioni di composizione con le classi `Screen`, `Keypad`, `CashDispenser` e `DepositSlot` ed è quindi sua la responsabilità della creazione di questi oggetti. La riga 25 crea un oggetto `BankDatabase` che permette alla classe `ATM` di manipolare le informazioni dei conti correnti bancari. [*Nota*: se questa fosse una macchina bancomat reale la classe `ATM` otterrebbe un riferimento ad un oggetto database esistente, creato dalla banca. In questa implementazione stiamo solo simulando il database della banca e quindi la classe `ATM` crea un oggetto database a suo uso e consumo.] Notate che le righe 6-10 includono le definizioni delle classi `Screen`, `Keypad`, `CashDispenser`, `DepositSlot` e `BankDatabase` in modo cha la classe `ATM` sia in grado di memorizzare oggetti di quelle classi.

Le righe 28-30 e 33 contengono i prototipi delle funzioni di utilità che la classe utilizza per realizzare i suoi compiti. Notate che la funzione membro `createTransaction` (riga 33) restituisce un puntatore alla classe `Transaction`. Per utilizzare il nome `Transaction` dobbiamo includere una dichiarazione anticipata di quella classe (riga 11). Si ricordi che una dichiarazione anticipata comunica al compilatore che una classe con quel nome esiste ma è definita altrove. Una dichiarazione anticipata è sufficiente in questo frangente perché viene solamente restituito un tipo `Transaction` mentre, se dovessimo creare un oggetto di quella classe, dovremmo includere l'intero file di intestazione della classe.

Definizione delle funzioni membro della classe ATM

La figura E.2 contiene le definizioni delle funzioni membro della classe `ATM`. Le righe 3-7 includono i file di intestazione necessari per l'implementazione del file `ATM.cpp`. Notate che l'inclusione del file di intestazione della classe `ATM` permette al compilatore di assicurarsi che tutte le funzioni membro della classe siano definite correttamente. Ciò permette inoltre alle funzioni membro di utilizzare i dati membro.

```
1    // ATM.cpp
2    // Definizione delle funzioni membro della classe ATM.
3    #include "ATM.h" // definizione della classe ATM
4    #include "Transaction.h" // definizione della classe Transaction
5    #include "BalanceInquiry.h" // definizione della classe BalanceInquiry
6    #include "Withdrawal.h" // definizione della classe Withdrawal
7    #include "Deposit.h" // definizione della classe Deposit
8
9    // costanti di enumerazione per le opzioni del menu
10   enum MenuOption { BALANCE_INQUIRY = 1, WITHDRAWAL, DEPOSIT, EXIT };
11
12   // costruttore di default della classe ATM
13   ATM::ATM()
```

Figura E.2 Definizione delle funzioni membro della classe ATM (continua)

```
14       : userAuthenticated ( false ), // all'inizio l'utente non è autenticato
15         currentAccountNumber( 0 ) // nessun conto corrente specificato
16    {
17       // corpo vuoto
18    } // fine del costruttore di default della classe ATM
19
20    // avvia l'applicazione ATM
21    void ATM::run()
22    {
23       // accoglie l'utente, lo autentica ed esegue le transazioni
24       while ( true )
25       {
26          // finché l'utente non è autenticato
27          while ( !userAuthenticated )
28          {
29             screen.displayMessageLine( "\nWelcome!" );
30             authenticateUser(); // autentica l'utente
31          } // fine del ciclo while
32
33          performTransactions(); // l'utente è adesso autenticato
34          userAuthenticated = false; // in preparazione della prossima sessione
35          currentAccountNumber = 0; // in preparazione della prossima sessione
36          screen.displayMessageLine( "\nThank you! Goodbye!" );
37       } // fine del ciclo while
38    } // fine della funzione run
39
40    // tentativo di autenticazione dell'utente
41    void ATM::authenticateUser()
42    {
43       screen.displayMessage( "\nPlease enter your account number: " );
44       int accountNumber = keypad.getInput(); // input del numero di conto
                                                   corrente
45       screen.displayMessage( "\nEnter your PIN: " ); // chiede il codice PIN
46       int pin = keypad.getInput(); // input del codice PIN
47
48       // imposta userAuthenticated al valore restituito dal database
49       userAuthenticated =
50          bankDatabase.authenticateUser( accountNumber, pin );
51
52       // verifica se l'utente è autenticato
53       if ( userAuthenticated )
54       {
55          currentAccountNumber = accountNumber; // memorizza il num. di conto
56       } // fine dell'if
57       else
58          screen.displayMessageLine(
59             "Invalid account number or PIN. Please try again." );
60    } // fine della funzione authenticateUser
61
```

Figura E.2 Definizione delle funzioni membro della classe ATM (continua)

```
62    // visualizza il menu principale ed esegue le transazioni
63    void ATM::performTransactions()
64    {
65       // puntatore locale alla transazione in atto
66       Transaction *currentTransactionPtr;
67
68       bool userExited = false; // l'utente non ha scelto di uscire
69
70       // cicla finché l'utente non sceglie di uscire
71       while ( !userExited )
72       {
73          // visualizza il menu principale e ottiene la scelta
74          int mainMenuSelection = displayMainMenu();
75
76          // decide come procedere in base alla scelta effettuata
77          switch ( mainMenuSelection )
78          {
79             // l'utente ha scelto una delle transazioni previste
80             case BALANCE_INQUIRY:
81             case WITHDRAWAL:
82             case DEPOSIT:
83                // inzializza un nuovo oggetto del tipo corretto
84                currentTransactionPtr =
85                   createTransaction( mainMenuSelection );
86
87                currentTransactionPtr->execute(); // esegue la transazione
88
89                // libera lo spazio di memoria dinamico della transazione
90                delete currentTransactionPtr;
91
92                break;
93             case EXIT: // l'utente ha scelto di terminare la sessione
94                screen.displayMessageLine( "\nExiting the system..." );
95                userExited = true; // la sessione termina
96                break;
97             default: // l'utente non ha inserito un valore corretto
98                screen.displayMessageLine(
99                   "\nYou did not enter a valid selection. Try again." );
100               break;
101         } // fine dello switch
102      } // fine del ciclo while
103   } // fine della funzione performTransactions
104
105   // visualizza il menu principale e ottiene la scelta dell'utente
106   int ATM::displayMainMenu() const
107   {
108      screen.displayMessageLine( "\nMain menu:" );
109      screen.displayMessageLine( "1 - View my balance" );
110      screen.displayMessageLine( "2 - Withdraw cash" );
```

Figura E.2 Definizione delle funzioni membro della classe ATM (continua)

```
111      screen.displayMessageLine( "3 - Deposit funds" );
112      screen.displayMessageLine( "4 - Exit\n" );
113      screen.displayMessage( "Enter a choice: " );
114      return keypad.getInput(); // restituisce la scelta dell'utente
115   } // fine della funzione displayMainMenu
116
117   // restituisce un oggetto della specifica classe derivata da Transaction
118   Transaction *ATM::createTransaction( int type )
119   {
120      Transaction *tempPtr; // puntatore temporaneo alla classe Transaction
121
122      // determina quale tipo di transazione creare
123      switch ( type )
124      {
125         case BALANCE_INQUIRY: // saldo del conto
126            tempPtr = new BalanceInquiry(
127               currentAccountNumber, screen, bankDatabase );
128            break;
129         case WITHDRAWAL: // prelievo
130            tempPtr = new Withdrawal( currentAccountNumber, screen,
131               bankDatabase, keypad, cashDispenser );
132            break;
133         case DEPOSIT: // deposito
134            tempPtr = new Deposit( currentAccountNumber, screen,
135               bankDatabase, keypad, depositSlot );
136            break;
137      } // fine dello switch
138
139      return tempPtr; // restituisce il nuovo oggetto creato
140   } // fine della funzione createTransaction
```

Figura E.2 Definizione delle funzioni membro della classe ATM

La riga 10 dichiara un dato `enum` di nome `MenuOption` che contiene le costanti necessarie a rappresentare le voci del menu principale (ovvero saldo del conto, prelievo, deposito e uscita). Osservate che l'impostazione di `BALANCE_INQUIRY` a `1` causa l'assegnamento alle successive costanti dei valori `2`, `3` e `4`, dato che le costanti di enumerazione vengono incrementate di `1`.

Le righe 13-18 definiscono il costruttore della classe che inizializza i dati membro. Quando viene inizialmente creato un oggetto `ATM` nessun utente è autenticato e la riga 14 utilizza quindi un inizializzatore di membro per impostare `userAuthenticated` a `false`. Allo stesso modo la riga 15 imposta `currentAccountNumber` a `0` perché non è stato ancora definito un numero di conto corrente sul quale agire.

La funzione membro `run` (righe 21-38) utilizza un ciclo infinito (righe 24-37) per ripetere la sequenza di operazioni dare il benvenuto ad un utente, autenticare l'utente e, se l'autenticazione va a buon fine, abilitare l'utente ad eseguire le transazioni. Dopo che un utente autenticato ha eseguito la sua transazione e decide di uscire la macchina ATM si resetta visualizzando un messaggio di arrivederci e ricominciando da capo. Viene utilizzato un ciclo infinito per simulare il fatto che la macchina sembra lavorare continuamente,

finché la banca non la spegne. Un utente può infatti uscire dal sistema ma non spegnere la macchina.

Nel ciclo infinito della funzione membro `run`, le righe 27-31ripetutamente danno il benvenuto ad un nuovo utente e tentano di autenticare l'utente finché un utente non viene autenticato (e l'attributo `userAuthenticated` viene impostato a `true`). La riga 29 invoca la funzione membro `displayMessageLine` della classe `Screen` per visualizzare un messaggio di benvenuto. Come l'analoga funzione membro `displayMessage` definita nei primi capitoli del libro, la funzione `displayMessageLine` (dichiarata alla riga 13 di figura E.3 e definita alle righe 20-23 di figura E.4) visualizza un messaggio sullo schermo ma, a differenza della precedente funzione, sposta il cursore su un a riga nuova. Abbiamo aggiunto questa funzione membro alla classe `Screen` durante l'implementazione per fornire un maggior controllo ai client sul posizionamento dei messaggi sullo schermo. La riga 30 di figura E.2 invoca la funzione d'utilità `authenticateUser` (righe 41-60) della classe `ATM` per cercare di autenticare l'utente corrente.

Per determinare i passi necessari ad autenticare un utente (prima di permettere qualunque tipo di transazione) bisogna fare riferimento alle specifiche del sistema. La riga 43 della funzione membro `authenticateUser` invoca la funzione membro `displayMessage` della classe `Screen` per chiedere all'utente di inserire il numero di conto corrente. La riga 44 invoca quindi la funzione membro `getInput` della classe `Keypad` per ottenere il valore inserito dall'utente, che viene memorizzato nella variabile locale `accountNumber`. La funzione membro `authenticateUser` chiede quindi all'utente di inserire il codice PIN (riga 45) e memorizza il valore inserito dall'utente nella variabile locale `pin` (riga 46). Le righe 49-50 tentano quindi di autenticare l'utente passando il numero di conto (`accountNumber`) e il codice PIN (`pin`) inseriti dall'utente alla funzione membro `authenticateUser` dell'oggetto `bankDatabase`. La classe `ATM` imposta il dato membro `userAuthenticated` al valore booleano restituito da questa funzione: se numero di conto e codice PIN corrispondono ai dati di un conto del database il valore assunto dal dato membro sarà `true` altrimenti rimarrà `false`. Se `userAuthenticated` diventa `true` la riga 55 memorizza il numero di conto corrente inserito dall'utente nel dato membro `currentAccountNumber`. Le altre funzioni membro della classe utilizzano questo dato qualora sia necessario accedere al numero di conto dell'utente. Se `userAuthenticated` è `false` le righe 58-59 utilizzano la funzione membro `displayMessageLine` della classe `Screen` per indicare che è stata inserita una coppia numero di conto / codice PIN errata e invita l'utente a riprovare. Notate che `currentAccountNumber` viene impostato soltanto dopo che il valore del conto corrente inserito dall'utente viene convalidato dal database.

Dopo che la funzione membro `run` tenta di autenticare un utente (riga 30) se il dato membro `userAuthenticated` è ancora `false` il ciclo `while` delle righe 27-31 viene eseguito di nuovo. Quando `userAuthenticated` diventa `true` il ciclo termina ed il controllo è trasferito alla riga 33, che chiama la funzione d'utilità `performTransactions`.

La funzione membro `performTransactions` (righe 63-103) realizza una sessione per l'utente autenticato. La riga 66 dichiara un puntatore locale di tipo `Transaction` che sarà collegato agli oggetti `BalanceInquiry`, `Withdrawal` e `Deposit` secondo necessità. Notate che si utilizza un puntatore alla classe `Transaction` per utilizzare proficuamente il polimorfismo. Notate inoltre che abbiamo usato il ruolo specificato nel diagramma delle classi di figura 3.20 (`currentTransaction`) per questo puntatore. Come di consueto, seguiamo la convenzione di aggiungere alla fine del nome delle variabili puntatore il suffisso "`Ptr`" ottenen-

do così il nome di variabile `currentTransactionPtr`. La riga 68 dichiara un'altra variabile locale `userExited` di tipo `bool` che viene impostata a `true` quando l'utente ha scelto di uscire. Questa variabile controlla il ciclo `while` (righe 71-102) che permette ad un utente di eseguire un numero di transazioni a piacere, prima di uscire. All'interno di questo ciclo la riga 74 visualizza il menu principale ed ottiene la scelta dell'utente, utilizzando la funzione d'utilità `displayMainMenu` (definita alle righe 106-115). Questa funzione membro visualizza il menu utilizzando le funzioni membro della classe `Screen` e restituisce il valore inserito dall'utente attraverso l'oggetto `keypad`. La funzione è dichiarata come `const` perché non modifica l'oggetto sul quale viene chiamata. La riga 74 memorizza la scelta dell'utente nella variabile locale `mainMenuSelection`.

Dopo aver ottenuto il valore selezionato dall'utente, la funzione `performTransactions` utilizza un'istruzione `switch` (righe 77-101) per agire di conseguenza. Se `mainMenuSelection` è uguale ad una delle costanti di enumerazione che rappresentano tipi di transazione, le righe 84-85 chiamano la funzione d'utilità `createTransaction` (definita alle righe 118-140) per restituire un puntatore ad un nuovo oggetto che rappresenta il tipo di transazione selezionato; tale puntatore viene assegnato alla variabile `currentTransactionPtr`. La riga 87 utilizza quindi quest'ultima variabile per invocare la funzione membro `execute` che esegue la transazione. Quando l'oggetto della classe derivata da `Transaction` non è più necessario viene infine distrutto e la memoria occupata rilasciata (riga 90).

Notate che alla variabile `currentTransactionPtr` viene assegnato l'indirizzo di un oggetto di una delle tre classi derivate da `Transaction`, in modo da eseguire le transazioni in modo polimorfo. Se l'utente sceglie di visualizzare il saldo del conto, ad esempio, la variabile `mainMenuSelection` sarà uguale a `BALANCE_INQUIRY` e `createTransaction` restituirà un puntatore ad un oggetto `BalanceInquiry`. La chiamata a `currentTransactionPtr->execute()` provoca quindi l'esecuzione della versione di `execute` della classe `BalanceInquiry`.

La funzione membro `createTransaction` (righe 118-140) utilizza inoltre un'istruzione `switch` (righe 123-137) per istanziare un nuovo oggetto derivato dalla classe `Transaction` in base al valore indicato dal parametro `type`. Notate che la funzione membro `performTransactions` passa `mainMenuSelection` a questa funzione membro solo quando `mainMenuSelection` contiene un valore corrispondente ad uno dei tre tipi di transazione previsti: `type` sarà quindi uguale a `BALANCE_INQUIRY` o `WITHDRAWAL` o `DEPOSIT`. Ogni `case` dell'istruzione `switch` collega il puntatore temporaneo `tempPtr` ad un nuovo oggetto del tipo di transazione prevista. Notate che ogni costruttore possiede la sua particolare lista di parametri, in base ai particolari dati, necessaria ad inizializzare quel tipo di oggetto. Un oggetto `BalanceInquiry` richiede solo il numero di conto corrente dell'utente e i riferimenti allo schermo ed al database (classi `Screen` e `BankDatabase`) mentre un oggetto `Withdrawal`, oltre agli stessi parametri, necessita anche di riferimenti al tastierino della macchina ATM e allo sportello dei prelievi (classi `Keypad` e `CashDispenser`) e un oggetto `Deposit` dei riferimenti al tastierino e allo sportello dei depositi (classi `Keypad` e `DepositSlot`). Notate che i costruttori delle classi che rappresentano le transazioni specificano dei parametri riferimento per ricevere gli oggetti che rappresentano le parti del sistema. Quando la funzione membro `createTransaction` infatti passa gli oggetti del sistema (`screen` e `keypad`) all'inizializzatore dei nuovi oggetti che vengono creati passa in realtà dei riferimenti agli oggetti esistenti. Discuteremo le classi che rappresentano le transazioni nelle sezioni E.9-E.12.

Dopo l'esecuzione di una transazione `userExited` rimane impostato a `false` e il ciclo `while` (righe 71-102) si ripete riportando l'utente al menu principale. Se l'utente seleziona l'opzione di uscita la riga 95 imposta `userExited` a `true` provocando l'uscita dal ciclo `while` perché la condizione di continuazione del ciclo (`!userExited`) diventa falsa. Il controllo viene quindi restituito alla funzione chiamante `run`. Se l'utente inserisce un'opzione di menu non valida (cioè un valore che non è compreso tra 1 e 4), le righe 98-99 visualizzano un apposito messaggio, `userExited` rimane impostato a `false` e l'utente è riportato al menu principale per una nuova scelta.

Quando `performTransactions` restituisce il controllo a `run` l'utente ha scelto di uscire dal sistema e le righe 34-35 resettano il sistema azzerando i dati membro `userAuthenticated` e `currentAccountNumber`, preparando il sistema per un nuovo utente. La riga 36 visualizza infine un messaggio di arrivederci prima che il sistema ricominci il suo ciclo da capo.

E.3 La classe Screen

La classe `Screen` (figure E.3-E.4) rappresenta lo schermo della macchina ATM e si occupa di tutti gli aspetti inerenti la visualizzazione di messaggi per l'utente. La classe `Screen` simula lo schermo della macchina ATM attraverso lo schermo del computer e visualizza i messaggi utilizzando lo stream `cout` e l'operatore di inserimento nello stream `<<`. Nei capitoli del libro la classe `Screen` possedeva una sola operazione, `displayMessage`, ma per una maggiore flessibilità nell'implementazione finale abbiamo incluso tre funzioni membro i cui prototipi appaiono alle righe 12-14 di figura E.3: `displayMessage`, `displayMessageLine` e `displayDollarAmount`.

Definizione delle funzioni membro della classe Screen

La figura E.4 contiene le definizioni delle funzioni membro della classe `Screen`. La riga 11 include la definizione della classe mentre la funzione membro `displayMessage` (righe 14-17) prende un argomento di tipo `string` e lo visualizza sulla console utilizzando `cout` e l'operatore `<<`. Il cursore rimane sulla stessa riga dell'output e quindi questa funzione è utile per visualizzare dei messaggi dai quali ci si attende una risposta dall'utente. Anche la funzione membro `displayMessageLine` (righe 20-23) visualizza un messaggio sulla console ma alla fine sposta il cursore su una riga nuova. La funzione `displayDollarAmount` (righe 26-29), infine, visualizza una valuta opportunamente formattata (ad esempio `$123.45`). La riga 28, infatti, utilizza i manipolatori di stream `fixed` e `setprecision` per visualizzare un valore numerico con due cifre decimali.

```
1    // Screen.h
2    // Definizione della classe Screen.
3    #ifndef SCREEN_H
4    #define SCREEN_H
5
6    #include <string>
7    using std::string;
8
9    class Screen
10   {
11   public:
```

Figura E.3 Definizione della classe Screen (continua)

```
12       void displayMessage( string ) const; // visualizza un messaggio
13       void displayMessageLine( string ) const; // visualizza un messaggio
                                                  e va a capo
14       void displayDollarAmount( double ) const; // visualizza una valuta
15    }; // fine della classe Screen
16
17    #endif // SCREEN_H
```

Figura E.3 Definizione della classe Screen

```
1     // Screen.cpp
2     // Definizione delle funzioni membro della classe Screen.
3     #include <iostream>
4     using std::cout;
5     using std::endl;
6     using std::fixed;
7
8     #include <iomanip>
9     using std::setprecision;
10
11    #include "Screen.h" // definizione della classe Screen
12
13    // visualizza un messaggio senza andare a capo
14    void Screen::displayMessage( string message ) const
15    {
16       cout << message;
17    } // fine della funzione displayMessage
18
19    // visualizza un messaggio e va a capo
20    void Screen::displayMessageLine( string message ) const
21    {
22       cout << message << endl;
23    } // fine della funzione displayMessageLine
24
25    // visualizza una valuta
26    void Screen::displayDollarAmount( double amount ) const
27    {
28       cout << fixed << setprecision( 2 ) << "$" << amount;
29    } // fine della funzione displayDollarAmount
```

Figura E.4 Definizione delle funzioni membro della classe Screen

E.4 La classe Keypad

La classe `Keypad` (figure E.5-E.6) rappresenta il tastierino della macchina ATM ed è responsabile della lettura dei dati inseriti dall'utente. Anche in questo caso il tastierino della macchina ATM viene simulato attraverso la tastiera del computer. Una tastiera normalmente prevede molti tasti che non compaiono sul tastierino della macchina ATM: supporremo, quindi, che l'utente digiti solo i caratteri corrispondenti al tastierino ATM ovvero i tasti corrispondenti alle cifre numeriche 0-9 e il tasto *Invio*. La riga 9 di figura E.5 contie-

ne il prototipo dell'unica funzione membro della classe `getInput`, dichiarata `const` perché non modifica l'oggetto sul quale viene chiamata.

```
1    // Keypad.h
2    // Definizione della classe Keypad.
3    #ifndef KEYPAD_H
4    #define KEYPAD_H
5
6    class Keypad
7    {
8    public:
9       int getInput() const; // restituisce il valore intero digitato dall'utente
10   }; // fine della classe Keypad
11
12   #endif // KEYPAD_H
```

Figura E.5 Definizione della classe Keypad

```
1    // Keypad.cpp
2    // Definizione delle funzioni membro della classe Keypad.
3    #include <iostream>
4    using std::cin;
5
6    #include "Keypad.h" // definizione della classe Keypad
7
8    // restituisce il valore intero digitato dall'utente
9    int Keypad::getInput() const
10   {
11      int input; // variabile per memorizzare il valore
12      cin >> input; // si assume che l'utente inserisca un valore intero
13      return input; // restituisce il valore inserito
14   } // fine della funzione getInput
```

Figura E.6 Definizione delle funzioni membro della classe Keypad

Definizione delle funzioni membro della classe Keypad

Nel file di implementazione della classe `Keypad` (figura E.6) la funzione membro `getInput` (definita alle righe 9-14) utilizza lo stream standard per l'input `cin` e l'operatore di estrazione dallo stream >> per ricevere l'input dell'utente. La riga 11 dichiara una variabile locale destinata a ricevere l'inserimento dell'utente, la riga 12 legge l'input nella variabile e la riga 13 restituisce il dato. La funzione `getInput` riceve tutto l'input per la macchina ATM, restituendo semplicemente il valore intero digitato dall'utente. Se un cliente della classe `Keypad` necessita di input che soddisfi qualche criterio (un valore corrispondente ad una delle possibili voci di un menu, ad esempio) deve farsi carico della convalida dei dati. [*Nota*: l'utilizzo dello stream standard per l'input `cin` e dell'operatore di estrazione dallo stream >> permette di leggere dall'utente qualunque tipo di input, non solo valori numerici. Siccome il tastierino reale della macchina ATM permette solo inserimenti numerici assumiamo che vengano inseriti solo dati numerici e non ci preoccupiamo dei casi in cui ciò non avviene.]

E.5 La classe CashDispenser

La classe `CashDispenser` (figure E.7-E.8) rappresenta lo sportello dei prelievi della macchina ATM. La definizione della classe (figura E.7) contiene il prototipo di un costruttore di default (riga 9) e dichiara altre due funzioni membro `public`: `dispenseCash` (riga 12) e `isSufficientCashAvailable` (riga 15). La classe conta sul fatto che una classe client (come ad esempio `Withdrawal`) chiami la funzione membro `dispenseCash` solo dopo aver verificato che vi sia disponibilità sufficiente di denaro (chiamando la funzione `isSufficientCashAvailable`). La funzione `dispenseCash` semplicemente simula l'uscita del denaro senza verificare se la disponibilità è sufficiente. La riga 17 dichiara la costante `private INITIAL_COUNT` che indica il numero iniziale di banconote presenti nello sportello dei prelievi (cioè `500`). La riga 18 implementa l'attributo `count` (modellato in figura 13.29) che tiene traccia del numero di banconote presenti nello sportello in un dato istante.

Definizione delle funzioni membro della classe CashDispenser

La figura E.8 contiene la definizione delle funzioni membro della classe `CashDispenser`. Il costruttore (righe 6-9) inizializza `count` al valore iniziale (cioè `500`) mentre la funzione membro `dispenseCash` simula l'emissione di denaro. Se il nostro sistema fosse realmente collegato ad un dispositivo di emissione di denaro dovrebbe interagire con esso nel rendere disponibili le banconote per l'utente. La nostra funzione invece si limita a decrementare il numero di banconote rimanenti del valore necessario a rendere disponile la cifra indicata da `amount` (riga 16). Notate infatti che la riga 16 calcola il numero di banconote da 20$ necessarie ad ottenere la cifra specificata da `amount`. Siccome l'ATM permette all'utente di scegliere solo importi di prelievo che sono multipli di 20, per ottenere il numero di banconote (`billsRequired`) necessarie è sufficiente dividere l'importo per 20. Osservate inoltre che è responsabilità del client della classe (`Withdrawal`) informare l'utente della disponibilità delle banconote: la classe `CashDispenser` non interagisce direttamente con lo schermo della macchina ATM.

La funzione membro `isSufficientCashAvailable` (righe 20-28) riceve un parametro `amount` che rappresenta la quantità di denaro da valutare e le righe 24-27 restituiscono `true` se `count` è maggiore o uguale a `billsRequired` (ovvero vi sono sufficienti banconote) e `false` altrimenti. Se ad esempio un utente volesse prelevare 80 $ (e quindi `billsRequired` sarà `4`) e rimangono solo tre banconote (cioè `count` è `3`), la funzione membro restituisce `false`.

```
1   // CashDispenser.h
2   // Definizione della classe CashDispenser.
3   #ifndef CASH_DISPENSER_H
4   #define CASH_DISPENSER_H
5
6   class CashDispenser
7   {
8   public:
9      CashDispenser(); // il costruttore inizializza il n. di banconote a 500
10
11     // simula l'erogazione di denaro
12     void dispenseCash( int );
13
```

Figura E.7 Definizione della classe CashDispenser (continua)

```
14       // indica se l'importo richiesto può essere erogato
15       bool isSufficientCashAvailable( int ) const;
16    private:
17       const static int INITIAL_COUNT = 500;
18       int count; // numero di banconote da $20 rimaste
19    }; // fine della classe CashDispenser
20
21    #endif // CASH_DISPENSER_H
```

Figura E.7 Definizione della classe CashDispenser

```
1     // CashDispenser.cpp
2     // Definizione delle funzioni membro della classe CashDispenser.
3     #include "CashDispenser.h" // definizione della classe CashDispenser
4
5     // costruttore di default
6     CashDispenser::CashDispenser()
7     {
8        count = INITIAL_COUNT; // imposta l'attributo count di default
9     } // fine del costruttore di default della classe CashDispenser
10
11    // simula l'erogazione di denaro; assume che vi sia disponibilità sufficiente
12    // (ovvero che una chiamata a isSufficientCashAvailable abbia restituito true)
13    void CashDispenser::dispenseCash( int amount )
14    {
15       int billsRequired = amount / 20; // numero di banconote da $20 necessarie
16       count -= billsRequired; // aggiorna il numero di banconote
17    } // fine della funzione dispenseCash
18
19    // indica se l'importo richiesto può essere erogato
20    bool CashDispenser::isSufficientCashAvailable( int amount ) const
21    {
22       int billsRequired = amount / 20; // numero di banconote da $20 necessarie
23
24       if ( count >= billsRequired )
25          return true; // banconote a sufficienza
26       else
27          return false; // numero di banconote insufficiente
28    } // fine della funzione isSufficientCashAvailable
```

Figura E.8 Definizione delle funzioni membro della classe CashDispenser

E.6 La classe DepositSlot

La classe `DepositSlot` (figure E.9-E.10) rappresenta lo sportello dei depositi della macchina ATM. Come nel caso della classe `CashDispenser` anche qui viene solo simulato il comportamento di un dispositivo hardware. La classe `DepositSlot` non possiede dati membro ed ha una sola funzione membro, `isEnvelopeReceived` (dichiarata alla riga 9 di figura E.9 e definita alle righe 7-10 di figura E.10), che indica se una busta con la valuta è stata inserita o meno.

```
1    // DepositSlot.h
2    // definizione della classe DepositSlot.
3    #ifndef DEPOSIT_SLOT_H
4    #define DEPOSIT_SLOT_H
5
6    class DepositSlot
7    {
8    public:
9       bool isEnvelopeReceived() const; // indica se la busta è stata ricevuta
10   }; // fine della classe DepositSlot
11
12   #endif // DEPOSIT_SLOT_H
```

Figura E.9 Definizione della classe DepositSlot

```
1    // DepositSlot.cpp
2    // Definizione delle funzioni membro della classe DepositSlot.
3    #include "DepositSlot.h" // definizione della classe DepositSlot
4
5    // indica se una busta è stata ricevuta (restituisce sempre true,
6    // perché questa è solo una simulazione di uno sportello reale)
7    bool DepositSlot::isEnvelopeReceived() const
8    {
9       return true; // busta ricevuta
10   } // fine della funzione isEnvelopeReceived
```

Figura E.10 Definizione delle funzioni membro della classe DepositSlot

Ricordate che nelle specifiche del sistema è stabilito in due minuti il tempo utile per inserire la busta di deposito nell'apposito sportello. Questa versione della funzione membro `isEnvelopeReceived` restituisce immediatamente `true` (riga 9 di figura E.10) assumendo che l'utente abbia inserito la busta in tempo utile: questa è infatti solo una simulazione software. Se il sistema fosse connesso ad una macchina reale la funzione `isEnvelopeReceived` avrebbe dovuto essere programmata in modo tale da attendere al massimo due minuti un segnale proveniente dal dispositivo di governo dello sportello dei depositi: se il segnale arriva nei due minuti previsti restituisce `true` altrimenti `false`.

E.7 La classe Account

La classe `Account` (figure E.11-E.12) rappresenta un conto corrente bancario. Le righe 9-15 della definizione della classe contengono i prototipi del costruttore e di altre sei funzioni membro. Ogni oggetto `Account` possiede quattro attributi (modellati in figura 13.29): `accountNumber`, `pin`, `availableBalance` e `totalBalance`. Le righe 17-20 implementano questi attributi come dati membro `private`. Il dato membro `availableBalance` rappresenta il totale del conto disponibile per i prelievi mentre il dato membro `totalBalance` rappresenta il saldo totale del conto dato dalla disponibilità per il prelievo più la quantità depositata in attesa di essere convalidata dalla banca.

Definizione delle funzioni membro della classe Account

La figura E.12 presenta le definizioni delle funzioni membro della classe `Account`. Il costruttore della classe (righe 6-14) prende come argomenti il numero di conto, il codice PIN, la disponibilità iniziale per i prelievi e quella totale.

La funzione membro `validatePIN` (righe 17-23) determina se il codice PIN specificato dall'utente (cioè il parametro `userPIN`) corrisponde al codice PIN associato al numero di conto (cioè il dato membro `pin`). Il parametro `userPIN` è stato modellato nel diagramma delle classi di figura 6.37. Se i due codici combaciano la funzione membro restituisce `true` (riga 20) altrimenti `false` (riga 22).

Le funzioni membro `getAvailableBalance` (righe 26-29) e `getTotalBalance` (righe 32-35) sono funzioni *get* per leggere rispettivamente i dati membro di tipo `double` `availableBalance` e `totalBalance`.

```
1    // Account.h
2    // Definizione della classe Account.
3    #ifndef ACCOUNT_H
4    #define ACCOUNT_H
5
6    class Account
7    {
8    public:
9       Account( int, int, double, double ); // costruttore per impostare gli attributi
10      bool validatePIN( int ) const; // il codice PIN specificato è corretto?
11      double getAvailableBalance() const; // restituisce il saldo effettivo
12      double getTotalBalance() const; // restituisce il saldo totale
13      void credit( double ); // accredita un importo sul conto
14      void debit( double ); // storna un importo dal conto
15      int getAccountNumber() const; // restituisce il numero di conto
16   private:
17      int accountNumber; // numero di conto corrente
18      int pin; // codice PIN per l'autenticazione
19      double availableBalance; // disponibilità per il prelievo
20      double totalBalance; // disponibilità per il prelievo +  denaro in attesa
                              di verifica
21   }; // fine della classe Account
22
23   #endif // ACCOUNT_H
```

Figura E.11 Definizione della classe Account

```
1    // Account.cpp
2    // Definizione delle funzioni membro della classe Account.
3    #include "Account.h" // definizione della classe Account
4
5    // costruttore
6    Account::Account( int theAccountNumber, int thePIN,
7       double theAvailableBalance, double theTotalBalance )
8       : accountNumber( theAccountNumber ),
```

Figura E.12 Definizione delle funzioni membro della classe Account (continua)

```
 9        pin( thePIN ),
10        availableBalance( theAvailableBalance ),
11        totalBalance( theTotalBalance )
12    {
13       // corpo vuoto
14    } // fine del costruttore della classe Account
15
16    // determina se il codice PIN spec. dall'utente coincide con quello mem.
         nel conto
17    bool Account::validatePIN( int userPIN ) const
18    {
19       if ( userPIN == pin )
20          return true;
21       else
22          return false;
23    } // fine della funzione validatePIN
24
25    // restituisce il saldo effettivo
26    double Account::getAvailableBalance() const
27    {
28       return availableBalance;
29    } // fine della funzione getAvailableBalance
30
31    // restituisce il saldo totale
32    double Account::getTotalBalance() const
33    {
34       return totalBalance;
35    } // fine della funzione getTotalBalance
36
37    // accredita un'importo sul conto
38    void Account::credit( double amount )
39    {
40       totalBalance += amount; // importo aggiunto al saldo totale
41    } // fine della funzione credit
42
43    // storna un'importo dal conto
44    void Account::debit( double amount )
45    {
46       availableBalance -= amount; // importo sottratto al saldo effettivo
47       totalBalance -= amount; // importo sottratto al saldo totale
48    } // fine della funzione debit
49
50    // restituisce il numero di conto corrente
51    int Account::getAccountNumber() const
52    {
53       return accountNumber;
54    } // fine della funzione getAccountNumber
```

Figura E.12 Definizione delle funzioni membro della classe Account

La funzione membro `credit` (righe 38-41) aggiunge una quantità di denaro (il parametro `amount`) ad un conto come parte di una transazione di deposito. Notate che l'importo viene sommato al solo dato membro `totalBalance` (riga 40) perché la somma depositata non è immediatamente disponibile per i prelievi ma lo diventa solo dopo che la banca ha verificato il contenuto della busta. L'implementazione della classe `Account` prevede le sole funzioni strettamente necessarie al funzionamento della macchina ATM e non sono quindi incluse le funzioni necessarie alla banca per aggiornare il valore di `availableBalance` (se il deposito viene accettato) o il valore di `totalBalance` (se il deposito è rifiutato).

La funzione membro `debit` (righe 44-48) sottrae dal conto una somma come parte di una transazione di prelievo. L'importo prelevato viene sottratto sia da `availableBalance` (riga 46) che da `totalBalance` (riga 47) perché un prelievo influisce su entrambe le quantità.

La funzione membro `getAccountNumber` (righe 51-54) fornisce l'accesso al dato membro `accountNumber`, per permettere ai client della classe (ad esempio la classe `BankDatabase`) di identificare il particolare conto in oggetto. Un oggetto `BankDatabase` infatti contiene diversi oggetti `Account` e può quindi invocare ripetutamente la funzione `getAccountNumber` su di essi per identificare un particolare conto.

E.8 La classe BankDatabase

La classe `BankDatabase` (figure E.13-E.14) rappresenta il database della banca con il quale la macchina ATM interagisce per modificare i dati di un conto corrente di un utente. La definizione della classe (figura E.13) dichiara i prototipi per un costruttore della classe e per diverse funzioni membro, oltre ai dati membro degli oggetti della classe. Uno di questi è stato ricavato dalla relazione di composizione con la classe `Account` che emerge in figura 13.28: un oggetto `BankDatabase` contiene zero o più oggetti `Account`. La riga 24 di figura E.13 implementa questa relazione dichiarando il dato membro `accounts` come `vector` di oggetti `Account`. Le righe 6-7 permettono il successivo uso di `vector`. La riga 27 contiene il prototipo della funzione d'utilità privata `getAccount` che permette alle altre funzioni membro della classe di ottenere un puntatore ad uno specifico oggetto `Account` all'interno di `accounts`.

```
1    // BankDatabase.h
2    // Definizione della classe BankDatabase.
3    #ifndef BANK_DATABASE_H
4    #define BANK_DATABASE_H
5
6    #include <vector> // utilizza vector per memorizzare gli oggetti Account
7    using std::vector;
8
9    #include "Account.h" // definizione della classe Account
10
11   class BankDatabase
12   {
13   public:
14      BankDatabase(); // il costruttore inizializza i conti correnti
15
16      // determina se numero di conto e codice PIN combaciano
```

Figura E.13 Definizione della classe BankDatabase (continua)

```
17       bool authenticateUser( int, int ); // restituisce true se il conto è autentico
18
19       double getAvailableBalance( int ); // ottiene il saldo effettivo
20       double getTotalBalance( int ); // ottiene il saldo totale
21       void credit( int, double ); // accredita un importo sul conto
22       void debit( int, double ); // storna un importo dal conto
23    private:
24       vector< Account > accounts; // vector dei conti correnti della banca
25
26       // funzione d'utilità private
27       Account * getAccount( int ); // get pointer to Account object
28    }; // fine della classe BankDatabase
29
30    #endif // BANK_DATABASE_H
```

Figura E.13 Definizione della classe BankDatabase

```
1     // BankDatabase.cpp
2     // Definizione delle funzioni membro della classe BankDatabase.
3     #include "BankDatabase.h" // definizione della classe BankDatabase
4
5     // Il costruttore di default inizializza i conti correnti
6     BankDatabase::BankDatabase()
7     {
8        // crea due oggetti Account per la verifica
9        Account account1( 12345, 54321, 1000.0, 1200.0 );
10       Account account2( 98765, 56789, 200.0, 200.0 );
11
12       // aggiunge i conti creati al vector accounts
13       accounts.push_back( account1 ); // aggiunge account1 alla fine del vector
14       accounts.push_back( account2 ); // aggiunge account2 alla fine del vector
15    } // fine del costruttore di default della classe BankDatabase
16
17    // recupera l'oggetto Account con il numero di conto specificato
18    Account * BankDatabase::getAccount( int accountNumber )
19    {
20       // scandisce gli elementi di accounts per trovare il conto
21       for ( size_t i = 0; i < accounts.size(); i++ )
22       {
23          // restituisce l'oggetto Account attuale se possiede quel numero di conto
24          if ( accounts[ i ].getAccountNumber() == accountNumber )
25             return &accounts[ i ];
26       } // fine del ciclo for
27
28       return NULL; // se non è stato trovato alcun conto, restituisce NULL
29    } // fine della funzione getAccount
30
31    // determina se il codice PIN specificato dall'utente coincide con
32    // quello memorizzato nel conto del database
33    bool BankDatabase::authenticateUser( int userAccountNumber,
```

Figura E.14 Definizione delle funzioni membro della classe BankDatabase (continua)

```
34       int userPIN )
35    {
36       // tentativo di recuperare il conto corrente con il num. specificato
37       Account * const userAccountPtr = getAccount( userAccountNumber );
38
39       // se il conto esiste restituisce il valore della funzione validatePIN
40       if ( userAccountPtr != NULL )
41          return userAccountPtr->validatePIN( userPIN );
42       else
43          return false; // numero di conto non trovato, restituisce false
44    } // fine della funzione authenticateUser
45
46    // restituisce il saldo effettivo del conto specificato
47    double BankDatabase::getAvailableBalance( int userAccountNumber )
48    {
49       Account * const userAccountPtr = getAccount( userAccountNumber );
50       return userAccountPtr->getAvailableBalance();
51    } // fine della funzione getAvailableBalance
52
53    // restituisce il saldo totale del conto specificato
54    double BankDatabase::getTotalBalance( int userAccountNumber )
55    {
56       Account * const userAccountPtr = getAccount( userAccountNumber );
57       return userAccountPtr->getTotalBalance();
58    } // fine della funzione getTotalBalance
59
60    // accredita un importo sul conto specificato
61    void BankDatabase::credit( int userAccountNumber, double amount )
62    {
63       Account * const userAccountPtr = getAccount( userAccountNumber );
64       userAccountPtr->credit( amount );
65    } // fine della funzione credit
66
67    // storna un importo dal conto specificato
68    void BankDatabase::debit( int userAccountNumber, double amount )
69    {
70       Account * const userAccountPtr = getAccount( userAccountNumber );
71       userAccountPtr->debit( amount );
72    } // fine della funzione debit
```

Figura E.14 Definizione delle funzioni membro della classe BankDatabase

Definizione delle funzioni membro della classe BankDatabase

La figura E.14 contiene le definizioni per le funzioni membro della classe `BankDatabase`. La classe è implementata con un costruttore di default (righe 6-15) che aggiunge oggetti `Account` al dato membro `accounts`. Per il solo scopo di verifica della classe, vengono creati due nuovi oggetti `Account` (righe 9-10) e inseriti alla fine dell'oggetto `vector` (righe 13-14). Notate che il costruttore degli oggetti `Account` prende quattro parametri: il numero di conto corrente, il codice PIN associato e i valori iniziali del saldo effettivo (`availableBalance`) e di quello totale (`totalBalance`).

Si ricordi che la classe `BankDatabase` funge da intermediario tra la classe `ATM` e gli oggetti `Account` che contengono l'informazione dei conti correnti. Le funzioni membro della classe `BankDatabase` non fanno quindi che richiamare le corrispondenti funzioni membro dell'oggetto `Account` che rappresenta il conto dell'utente corrente della macchina ATM.

La funzione d'utilità `getAccount` (riga 18-29) permette di ottenere un puntatore ad un particolare oggetto `Account` all'interno di `accounts`. Per localizzare l'oggetto `Account`, si confronta ripetutamente il valore restituito dalla funzione membro `getAccountNumber` di ogni elemento di `accounts` con il valore da cercare. Le righe 21-26 effettuano la scansione di tutti gli elementi di `accounts`. Se il numero di conto corrente dell'oggetto attuale (cioè `accounts[i]`) è uguale al parametro `accountNumber`, la funzione restituisce immediatamente l'indirizzo dell'oggetto. Se alla fine nessun elemento di `accounts` possiede il numero cercato al funzione restituisce `NULL` (riga 28). Notate che la funzione deve restituire un puntatore piuttosto che un riferimento perché il valore restituito può anche essere `NULL` e un riferimento non può assumere tale valore (un puntatore invece sì).

Notate inoltre che la funzione membro `size` di `vector` (invocata alla riga 21 nella condizione di continuazione del ciclo) restituisce il numero di elementi di `vector` come dato di tipo `size_t` (che di solito corrisponde al tipo `unsigned int`). Per tale motivo la variabile di controllo del ciclo `i` è dichiarata di tipo `size_t`. Con alcuni compilatori, infatti, dichiarare `i` come `int` causa un avvertimento da parte del compilatore perché nella condizione di continuazione del ciclo si confrontano un valore con segno (di tipo `int`) con un valore senza segno (di tipo `size_t`).

La funzione membro `authenticateUser` (righe 33-44) verifica l'identità dell'utente della macchina ATM. Questa funzione riceve come argomenti il numero di conto corrente e il relativo codice PIN specificati dall'utente e verifica se la coppia corrisponde ad un conto corrente del database. La riga 37 chiama la funzione di utilità `getAccount` che restituisce o un puntatore all'oggetto `Account` che ha numero `userAccountNumber` o `NULL` se il numero di conto specificato non è valido. Abbiamo dichiarato `userAccountNumber` come `const` perché quando il puntatore è collegato al relativo oggetto non viene più modificato, per tutta la sua durata in vita. Se `getAccount` restituisce un puntatore ad un oggetto `Account`, la riga 41 restituisce il valore booleano della funzione membro `validatePIN`. Notate che la funzione membro `authenticateUser` della classe `BankDatabase` non effettua direttamente il controllo del codice PIN ma passa il parametro `userPIN` alla funzione membro `validatePIN` della classe `Account`. Il valore restituito da quest'ultima indica se il codice PIN specificato dall'utente corrisponde a quello memorizzato per il conto corrente e la funzione `authenticateUser` restituisce il valore al client della classe (la classe `ATM`).

La classe `BankDatabase` conta sul fatto che la classe `ATM`, prima di effettuare delle transazioni, chiami la funzione membro `authenticateUser` e che questa restituisca il valore `true`. Viene inoltre dato per scontato che in tutte le transazioni che seguono la classe `ATM` utilizzi il numero di conto corrente specificato durante l'autenticazione e che questo sia il contenuto dell'argomento `userAccountNumber` passato alle funzioni membro di `BankDatabase`. Le funzioni membro `getAvailableBalance` (righe 47-52), `getTotalBalance` (righe 54-58), `credit` (righe 61-65) e `debit` (righe 68-72) semplicemente recuperano un puntatore all'oggetto `Account`, specificato da `userAccountNumber`, con la funzione di utilità `getAccount` e richiamano quindi le corrispondenti funzioni membro della classe `Account`. Sappiamo che le chiamate a `getAccount` non restituiranno mai il valore `NULL` perché `userAccountNumber` si riferisce ad un conto realmente esisten-

te. Osservate che `getAvailableBalance` e `getTotalBalance` restituiscono semplicemente i valori ottenuti dalle corrispondenti funzioni della classe `Account`. Le funzioni `credit` e `debit` semplicemente inoltrano il parametro `amount` alle funzioni della classe `Account` che invocano.

E.9 La classe *Transaction*

La classe `Transaction` (figure E.15-E.16) è una classe base astratta che rappresenta la nozione di transazione finanziaria per la macchina ATM e contiene le caratteristiche comuni delle classi derivate `BalanceInquiry`, `Withdrawal` e `Deposit`. La figura E.15 costituisce un'espansione del file di intestazione introdotto nella sezione 13.10. Le righe 13, 17-19 e 22 contengono i prototipi del costruttore e di quattro funzioni membro. La riga 15 definisce un distruttore virtuale: in tal modo tutti i distruttori delle classi derivate sono virtuali (anche quelli definiti implicitamente dal compilatore) e ciò assicura che gli oggetti delle classi derivate allocati dinamicamente vengano distrutti nel modo appropriato anche quando vengono eliminati attraverso un puntatore alla classe base. Le righe 24-26 dichiarano i dati membro `private` della classe. Dalla figura 13.29 si può vedere che la classe `Transaction` contiene un attributo `accountNumber` (implementato alla riga 24) che indica il conto corrente utilizzato per la transazione. I dati membro `screen` (riga 25) e `bankDatabase` (riga 26) vengono ricavati dalle associazioni rappresentate in figura 13.28: tutte le transazioni necessitano di accedere allo schermo della macchina ATM e al database della banca. La classe `Transaction` include quindi i riferimenti a oggetti di classe `Screen` e `BankDatabase`, che vengono inizializzati nel costruttore. Osservate le dichiarazioni anticipate alle righe 6-7: stanno a significare che il file di intestazione contiene riferimenti a oggetti delle classi `Screen` e `BankDatabase` ma che queste classi sono definite altrove.

```
1    // Transaction.h
2    // Definizione della casse base astratta Transaction.
3    #ifndef TRANSACTION_H
4    #define TRANSACTION_H
5
6    class Screen; // dichiarazione anticipata della classe Screen
7    class BankDatabase; // dichiarazione anticipata della classe BankDatabase
8
9    class Transaction
10   {
11   public:
12      // il costruttore inizializza le caratteristiche comuni di tutte le transazioni
13      Transaction( int, Screen &, BankDatabase & );
14
15      virtual ~Transaction() { } // distruttore virtuale con corpo vuoto
16
17      int getAccountNumber() const; // restituisce il numero di conto
18      Screen &getScreen() const; // restituisce un riferimento allo schermo
19      BankDatabase &getBankDatabase() const; // restituisce un riferimento
                                               al database
20
21      // funzione virtuale pura per realizzare la transazione
```

Figura E.15 Definizione della classe Transaction (continua)

```
22       virtual void execute() = 0; // riscritta nelle classi derivate
23    private:
24       int accountNumber; // indica il conto corrente oggetto della transazione
25       Screen &screen; // riferimento allo schermo dell'ATM
26       BankDatabase &bankDatabase; // riferimento al database della banca
27    }; // fine della classe Transaction
28
29    #endif // TRANSACTION_H
```

Figura E.15 Definizione della classe Transaction

```
1     // Transaction.cpp
2     // Definizione delle funzioni membro della classe Transaction.
3     #include "Transaction.h" // definizione della classe Transaction
4     #include "Screen.h" // definizione della classe Screen
5     #include "BankDatabase.h" // definizione della classe BankDatabase
6
7     // il costruttore inizializza le caratteristiche comuni di tutte le transazioni
8     Transaction::Transaction( int userAccountNumber, Screen &atmScreen,
9        BankDatabase &atmBankDatabase )
10       : accountNumber( userAccountNumber ),
11         screen( atmScreen ),
12         bankDatabase( atmBankDatabase )
13    {
14       // corpo vuoto
15    } // fine del costruttore della classe Transaction
16
17    // restituisce il numero di conto corrente
18    int Transaction::getAccountNumber() const
19    {
20       return accountNumber;
21    } // fine della funzione getAccountNumber
22
23    // restituisce il riferimento allo schermo
24    Screen &Transaction::getScreen() const
25    {
26       return screen;
27    } // fine della funzione getScreen
28
29    // restituisce il riferimento al database della banca
30    BankDatabase &Transaction::getBankDatabase() const
31    {
32       return bankDatabase;
33    } // fine della funzione getBankDatabase
```

Figura E.16 Definizione delle funzioni membro della classe Transaction (continua)

La classe `Transaction` possiede un costruttore (dichiarato alla riga 13 di figura E.15 e definito alle righe 8-15 di figura E.16) che prende come argomenti il numero del conto corrente dell'utente attuale e i riferimenti allo schermo della macchina ATM e al database della banca. Dato che la classe `Transaction` è astratta questo costruttore non verrà mai

chiamato direttamente per inizializzare nuovi oggetti della classe ma sarà chiamato attraverso la sintassi degli inizializzatori della classe base dai costruttori delle classi derivate.

La classe `Transaction` prevede tre funzioni *get* pubbliche: `getAccountNumber` (dichiarata alla riga 17 di figura E.15 e definita alle righe 18-21 di figura E.16), `getScreen` (dichiarata alla riga 18 di figura E.15 e definita alle righe 24-27 di figura E.16) e `getBankDatabase` (dichiarata alla riga 19 di figura E.15 e definita alle righe 30-33 di figura E.16). Le classi derivate ereditano queste funzioni per mezzo delle quali possono accedere ai dati membro privati della classe base.

La classe `Transaction` dichiara inoltre una funzione virtuale pura `execute` (riga 22 di figura E.15): non ha senso infatti fornire un'implementazione di questa funzione membro qui perché una generica transazione non viene mai eseguita. La funzione viene quindi dichiarata virtuale pura in modo da forzare le classi derivate a fornire un'appropriata implementazione, adatta alla particolare transazione che rappresentano.

E.10 La classe BalanceInquiry

La classe `BalanceInquiry` (figure E.17-E.18) è derivata dalla classe base astratta `Transaction` e rappresenta la transazione di interrogazione del saldo di un conto. La classe `BalanceInquiry` non possiede dati membro propri ma eredita i dati membro della classe base `accountNumber`, `screen` e `bankDatabase`, accessibili attraverso le funzioni *get* pubbliche della classe base. Notate che la riga 6 include la definizione della classe base `Transaction`. Il costruttore della classe `BalanceInquiry` (dichiarato alla riga 11 di figura E.17 e definito alle righe 8-13 di figura E.18) riceve gli argomenti corrispondenti ai dati membro della classe base e li passa al costruttore della classe base, utilizzando la sintassi degli inizializzatori (riga 10 di figura E.18). La riga 12 di figura E.17 contiene il prototipo della funzione membro `execute`, richiesto per segnalare l'intenzione di riscrivere la funzione membro virtuale pura della classe base con lo stesso nome.

La classe `BalanceInquiry` riscrive infatti la funzione virtuale pura `execute` della classe `Transaction` per fornire un'implementazione concreta (righe 16-37 di figura E.18) che esegua i passi necessari a realizzare l'interrogazione del saldo del conto. Le righe 19-20 impostano i riferimenti al database della banca e allo schermo della macchina ATM richiamando le funzioni membro ereditate dalla classe base. Le righe 23-24 ottengono il saldo effettivo del conto utilizzando la funzione `getAvailableBalance` della classe `BankDatabase`. Notate che la riga 24 utilizza la funzione membro ereditata `getAccountNumber` per ottenere il numero di conto corrente dell'utente che sta usando la macchina ATM e passarlo alla funzione `getAvailableBalance`. Le righe 27-28 ottengono invece il saldo totale del conto dell'utente autenticato. Le righe 31-36 visualizzano il valore del saldo sullo schermo della macchina ATM. Ricordate che la funzione `displayDollarAmount` prende una argomento di tipo `double` e lo visualizza in un formato opportuno. Se il saldo effettivo di un utente è `700.5`, ad esempio, la riga 33 visualizzerà `$700.50`. Notate che la riga 36 inserisce una linea vuota sullo schermo per separare il saldo del conto dall'output seguente (ovvero il menu principale visualizzato dalla classe `ATM` dopo l'esecuzione della transazione).

```
1    // BalanceInquiry.h
2    // Definizione della classe BalanceInquiry.
3    #ifndef BALANCE_INQUIRY_H
4    #define BALANCE_INQUIRY_H
```

Figura E.17 Definizione della classe BalanceInquiry (continua)

```
5
6     #include "Transaction.h" // definizione dela classe Transaction
7
8     class BalanceInquiry : public Transaction
9     {
10    public:
11       BalanceInquiry( int, Screen &, BankDatabase & ); // costruttore
12       virtual void execute(); // esegue la transazione
13    }; // fine della classe BalanceInquiry
14
15    #endif // BALANCE_INQUIRY_H
```

Figura E.17 Definizione della classe BalanceInquiry

```
1     // BalanceInquiry.cpp
2     // Definizione delle funzioni membro della classe BalanceInquiry.
3     #include "BalanceInquiry.h" // definizione della classe BalanceInquiry
4     #include "Screen.h" // definizione della classe Screen
5     #include "BankDatabase.h" // definizione della classe BankDatabase
6
7     // il costruttore inizializza i dati membro della classe base
8     BalanceInquiry:: BalanceInquiry( int userAccountNumber, Screen &atmScreen,
9        BankDatabase &atmBankDatabase )
10       : Transaction( userAccountNumber, atmScreen, atmBankDatabase )
11    {
12       // corpo vuoto
13    } // fine del costruttore della classe BalanceInquiry
14
15    // esegue la transazione; riscrive la funzione virtuale pura della classe base
16    void BalanceInquiry::execute()
17    {
18       // ottiene i riferimenti allo schermo e al database
19       BankDatabase &bankDatabase = getBankDatabase();
20       Screen &screen = getScreen();
21
22       // ottiene il saldo effettivo del conto dell'utente corrente
23       double availableBalance =
24          bankDatabase.getAvailableBalance( getAccountNumber() );
25
26       // ottiene il saldo totale del conto dell'utente corrente
27       double totalBalance =
28          bankDatabase.getTotalBalance( getAccountNumber() );
29
30       // visualizza le informazioni del saldo sullo schermo
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    } // fine della funzione execute
```

Figura E.18 Definizione delle funzioni membro della classe BalanceInquiry

E.11 La classe Withdrawal

La classe `Withdrawal` (figure E.19-E.20) rappresenta una transazione per un prelievo. La figura E.19 espande il file di intestazione sviluppato in figura 13.31. La classe `Withdrawal` possiede un costruttore e una funzione membro `execute`. Ricordate che in figura 13.29 viene indicato che la classe `Withdrawal` possiede un attributo `amount`, che la riga 16 implementa come dato intero. La figura 13.28 modella le associazioni della classe `Withdrawal` con le classi `Keypad` e `CashDispenser` per cui le righe 17-18 dichiarano i riferimenti `keypad` e `cashDispenser`. La riga 19 costituisce invece il prototipo di una funzione di utilità.

```
1    // Withdrawal.h
2    // Definizione della classe Withdrawal.
3    #ifndef WITHDRAWAL_H
4    #define WITHDRAWAL_H
5
6    #include "Transaction.h" // definizione della classe Transaction
7    class Keypad; // dichiarazione anticipata della classe Keypad
8    class CashDispenser; // dichiarazione anticipata della classe CashDispenser
9
10   class Withdrawal : public Transaction
11   {
12   public:
13      Withdrawal( int, Screen &, BankDatabase &, Keypad &, CashDispenser & );
14      virtual void execute(); // esegue la transazione
15   private:
16      int amount; // importo da prelevare
17      Keypad &keypad; // riferimento al tastierino dell'ATM
18      CashDispenser &cashDispenser; // riferimento allo sportello dei prelievi
19      int displayMenuOfAmounts() const; // visualizza il menu dei prelievi
20   }; // fine della classe Withdrawal
21
22   #endif // WITHDRAWAL_H
```

Figura E.19 Definizione della classe Withdrawal

```
1    // Withdrawal.cpp
2    // Definizione delle funzioni membro della classe Withdrawal.
3    #include "Withdrawal.h" // definizione della classe Withdrawal
4    #include "Screen.h" // definizione della classe Screen
5    #include "BankDatabase.h" // definizione della classe BankDatabase
6    #include "Keypad.h" // definizione della classe Keypad
7    #include "CashDispenser.h" // definizione della classe CashDispenser
8
9    // costante globale che corrisponde all'opzione di annullamento
10   const static int CANCELED = 6;
11
12   // il costruttore inizializza i dati membro
13   Withdrawal::Withdrawal( int userAccountNumber, Screen &atmScreen,
14      BankDatabase &atmBankDatabase, Keypad &atmKeypad,
```

Figura E.20 Definizione delle funzioni membro della classe Withdrawal (continua)

```
15       CashDispenser &atmCashDispenser )
16       : Transaction( userAccountNumber, atmScreen, atmBankDatabase ),
17         keypad( atmKeypad ), cashDispenser( atmCashDispenser )
18    {
19       // corpo vuoto
20    } // fine del costruttore della classe Withdrawal
21
22    // esegue la transazione; riscrive la funzione virtuale pura della classe base
23    void Withdrawal::execute()
24    {
25       bool cashDispensed = false; // denaro non ancora erogato
26       bool transactionCanceled = false; // transazione non annullata
27
28       // ottiene i riferimenti allo schermo e al database
29       BankDatabase &bankDatabase = getBankDatabase();
30       Screen &screen = getScreen();
31
32       // cicla finché il denaro viene erogato o la transazione è cancellata
33       do
34       {
35          // ottiene l'importo selezionato dall'utente
36          int selection = displayMenuOfAmounts();
37
38          // verifica se l'utente ha selezionato un importo
39          if ( selection != CANCELED )
40           {
41             amount = selection; // imposta amount all'importo scelto
42
43             // ottiene il saldo effettivo del conto in oggetto
44             double availableBalance =
45                bankDatabase.getAvailableBalance( getAccountNumber() );
46
47             // verifica la disponibilità dell'utente
48             if ( amount <= availableBalance )
49              {
50                // verifica se c'è disponibilità sufficiente nello sportello
51                if ( cashDispenser.isSufficientCashAvailable( amount ) )
52                 {
53                   // aggiorna il conto dell'utente
54                   bankDatabase.debit( getAccountNumber(), amount );
55
56                   cashDispenser.dispenseCash( amount ); // eroga il denaro
57                   cashDispensed = true; // denaro erogato
58
59                   // ricorda all'utente di prelevare il denaro
60                   screen.displayMessageLine(
61                      "\nPlease take your cash from the cash dispenser." );
62                } // fine dell'if
63                else // non vi sono banconote sufficienti
```

Figura E.20 Definizione delle funzioni membro della classe Withdrawal (continua)

```
64                screen.displayMessageLine(
65                   "\nInsufficient cash available in the ATM."
66                   "\n\nPlease choose a smaller amount." );
67             } // fine dell'if
68             else // disponibilità dell'utente insufficiente
69              {
70                screen.displayMessageLine(
71                   "\nInsufficient funds in your account."
72                   "\n\nPlease choose a smaller amount." );
73             } // fine dell'else
74          } // fine dell'if
75          else // annullamento dell'operazione
76           {
77             screen.displayMessageLine( "\nCanceling transaction..." );
78             transactionCanceled = true; // user canceled the transaction
79          } // fine dell'else
80       } while ( !cashDispensed && !transactionCanceled ); // fine del ciclo do...while
81    } // fine della funzione execute
82
83    // visualizza il menu dei prelievi; restituisce l'importo
84    // selezionato o 0 se l'utente ha scelto l'annullamento
85    int Withdrawal::displayMenuOfAmounts() const
86    {
87       int userChoice = 0; // variabile locale per memorizzare la scelta dell'utente
88
89       Screen &screen = getScreen(); // ottiene il riferimento allo schermo
90
91       // array degli importi che è possibile prlelevare
92       int amounts[] = { 0, 20, 40, 60, 100, 200 };
93
94       // cicla finché non viene effettuata una scelta corretta
95       while ( userChoice == 0 )
96       {
97          // visualizza il 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(); // ottiene la scelta dell'utente
108
109         // determina come procedere in base alla scelta effettuata
110         switch ( input )
111          {
112            case 1: // importo da prelevare valido
```

Figura E.20 Definizione delle funzioni membro della classe Withdrawal (continua)

```
113           case 2: // (cioè le opzioni 1, 2, 3, 4 o 5), restituisce il
114           case 3: // corrispondente importo dall'array amounts
115           case 4:
116           case 5:
117             userChoice = amounts[ input ]; // memorizza la scelta dell'utente
118             break;
119           case CANCELED: // annullamento dell'operazione
120             userChoice = CANCELED; // memorizza la scelta dell'utente
121             break;
122           default: // valore inserito scoretto
123             screen.displayMessageLine(
124                "\nIvalid selection. Try again." );
125         } // fine dello switch
126       } // fine del ciclo while
127
128       return userChoice; // restituisce l'importo o la costante CANCELED
129     } // fine della funzione displayMenuOfAmounts
```

Figura E.20 Definizione delle funzioni membro della classe Withdrawal

Definizione delle funzioni membro della classe Withdrawal

La figura E.20 contiene le definizioni delle funzioni membro della classe `Withdrawal`. La riga 3 include la definizione della classe mentre le righe 4-7 includono le definizioni delle altre classi utilizzate dalle funzioni membro della classe. La riga 11 dichiara una costante globale corrispondente all'opzione di cancellazione del menu di prelievo.

Il costruttore della classe `Withdrawal` (definito alle righe 13-20 di figura E.20) prende cinque parametri e utilizza la sintassi degli inizializzatori della classe base (riga 16) per passare i parametri `userAccountNumber`, `atmScreen` e `atmBankDatabase` al costruttore della classe base, per inizializzare i dati membro ereditati. Il costruttore assegna inoltre i parametri `atmKeypad` e `atmCashDispenser` ai dati membro `keypad` e `cashDispenser` utilizzando gli inizializzatori di membro (riga 17).

La classe `Withdrawal` riscrive inoltre la funzione virtuale pura `execute` con un'implementazione concreta (righe 23-81) che esegue i passi necessari a portare a termine una transazione di tipo prelievo. La riga 25 dichiara e inizializza una variabile locale booleana `cashDispensed` che serve a indicare se il denaro è stato emesso (se la transazione si è cioè conclusa con successo) e inizialmente è impostata a `false`. La riga 26 dichiara e inizializza a `false` la variabile booleana `transactionCanceled` che indica se la transazione è stata cancellata dall'utente. Le righe 29-30 ottengono i riferimenti al database della banca e allo schermo della macchina ATM invocando le funzioni membro ereditate dalla classe base.

Le righe 33-80 contengono un ciclo `do…while` che viene eseguito finché il denaro è emesso (ovvero finché `cashDispensed` diventa `true`) o l'utente ha cancellato la transazione (ovvero `transactionCanceled` è diventata `true`). Questo ciclo è utilizzato per riportare continuamente l'utente all'inizio della transazione se si verifica qualche errore (ad esempio l'importo richiesto è maggiore della disponibilità del conto dell'utente o della quantità di denaro presente nella macchina ATM). La riga 36 visualizza il menu dei prelievi e ottiene la scelta dell'utente chiamando la funzione privata di utilità `displayMenuOfAmounts` (definita alle righe 85-129). Questa funzione membro visualizza il menu degli importi

selezionabili e restituisce un intero corrispondente alla scelta dell'utente o la costante intera `CANCELED`, che indica che l'utente ha scelto di annullare la transazione.

La funzione membro `displayMenuOfAmounts` (righe 85-129) come prima cosa dichiara la variabile locale `userChoice` (inizialmente impostata a `0`) che memorizza il valore restituito della funzione (riga 87). La riga 89 ottiene un riferimento allo schermo della macchina ATM invocando la funzione membro `getScreen` ereditata dalla classe `Transaction`. La riga 92 dichiara un array di valori interi corrispondenti agli importi che è possibile prelevare e che vengono visualizzati nel menu dei prelievi. Il primo elemento dell'array (quello di indice `0`) viene ignorato perché non vi è un'opzione nel menu con tale valore. Il ciclo `while` delle righe 95-126 viene ripetuto fintanto che la variabile `userChoice` non assume un valore diverso da `0`, ovvero fino a quando l'utente non opera una scelta corretta tra le opzioni del menu. Le righe 98-105 visualizzano il menu dei prelievi e chiedono all'utente di operare una scelta che viene letta alla riga 107 nella variabile `input`. L'istruzione `switch` delle righe 110-125 decide come procedere in base alla scelta operata dall'utente. Se l'utente ha inserito un valore intero compreso tra `1` e `5` la riga 117 imposta la variabile `userChoice` al valore di `amounts` di indice `input`. Se ad esempio l'utente ha inserito il valore `3` per prelevare `$60` allora la riga 117 imposta `userChoice` al valore di `amounts[3]` (cioè `60`). La riga 118 termina l'istruzione `switch`. La variabile `userChoice` a questo punto non è più `0` e il ciclo `while` delle righe 95-126 termina e la riga 128 restituisce `userChoice`. Se l'utente seleziona l'opzione di annullamento vengono eseguite le righe 120-121 che impostano la variabile `userChoice` a `CANCELED` e la funzione membro termina restituendo questo valore. Se infine l'utente non inserisce un valore valido le righe 123-124 visualizzano un messaggio di errore e viene riproposto il menu dei prelievi.

L'istruzione `if` alla riga 39 della funzione membro `execute` determina se l'utente ha scelto un certo importo da prelevare o se ha annullato l'operazione. Nel secondo caso vengono eseguite le righe 77-78 che visualizzano un messaggio appropriato e impostano `transactionCanceled` a `true`. Ciò rende falsa la condizione di continuazione del ciclo alla riga 80 e il controllo ritorna alla funzione membro chiamante (ovvero la funzione `performTransactions` della classe `ATM`). Se invece l'utente ha scelto un certo importo la riga 41 assegna la variabile locale `selection` al dato membro `amount`. Le righe 44-45 leggono il saldo effettivo del conto dell'utente corrente e lo memorizzano nella variabile locale `availableBalance` di tipo `double`. L'istruzione `if` alla riga 48 determina se l'importo selezionato è minore o uguale al saldo effettivo del conto. Se non lo è le righe 70-72 visualizzano un apposito messaggio di errore e il controllo è trasferito alla fine del ciclo `do…while` e il ciclo si ripete perché entrambe `cashDispensed` e `transactionCanceled` sono ancora impostate a `false`. Se il saldo del conto è sufficiente l'istruzione `if` alla riga 51 determina se vi sono nello sportello sufficienti banconote per soddisfare la richiesta chiamando la funzione membro `isSufficientCashAvailable` dell'oggetto `cashDispenser`. Se questa funzione restituisce il valore `false` allora le righe 64-66 visualizzano un apposito messaggio di errore e il ciclo `do…while` riparte da capo. Se invece la disponibilità di banconote è sufficiente allora i requisiti per il prelievo sono tutti soddisfatti e la riga 54 storna il valore `amount` dal conto dell'utente nel database. Le righe 56-57 rendono quindi disponibile il denaro per l'utente e impostano `cashDispensed` al valore `true`. Le righe 60-61, infine, visualizzano un messaggio che indica che il denaro è stato erogato. Siccome `cashDispensed` è adesso impostato a `true`, il controllo è trasferito all'istruzione successiva al ciclo `do…while`. Dato che non vi sono istruzioni dopo il ciclo il controllo viene restituito alla classe `ATM`.

Notate che nelle chiamate di funzione alle righe 64-66 e 70-72 l'argomento della funzione membro `displayMessageLine` della classe `Screen` è diviso in due stringhe letterali, ognuna delle quali compare su una riga diversa del programma. Questa divisione è dettata dal fatto che l'argomento è troppo lungo per stare su una sola riga. Il C++ concatena (cioè unisce) due stringhe adiacenti in un programma, anche se si trovano su righe separate. Se scrivete, ad esempio, in un programma `"Happy " "Birthday"` il C++ vede queste due stringhe letterali adiacenti come l'unica stringa `"Happy Birthday"`. Quando vengono eseguite le righe 64-66, la funzione `displayMessageLine` riceve quindi un unico parametro di tipo `string`, anche se l'argomento è spezzato in due stringhe letterali.

E.12 La classe Deposit

La classe `Deposit` (figure E.21-E.22) è derivata dalla classe `Transaction` e rappresenta una transazione di tipo deposito di valuta. La figura E.21 contiene la definizione della classe. Come le altre classi derivate da `Transaction` `BalanceInquiry` e `Withdrawal` anche `Deposit` dichiara un costruttore (riga 13) e una funzione membro `execute` (riga 14). Ricordate dal diagramma delle classi di figura 13.29 che la classe `Deposit` possiede un attributo `amount` che è implementato alla riga 16 con un dato membro intero. Le righe 17-18 creano i dati membro riferimento `keypad` e `depositSlot` che implementano le associazioni della classe `Deposit` con le classi `Keypad` e `DepositSlot`, rappresentate in figura 13.28. La riga 19 contiene il prototipo della funzione di utilità privata `promptForDepositAmount`.

```
1    // Deposit.h
2    // Definizione della classe Deposit.
3    #ifndef DEPOSIT_H
4    #define DEPOSIT_H
5
6    #include "Transaction.h" // definizione della classe Transaction
7    class Keypad; // dichiarazione anticipata della classe Keypad
8    class DepositSlot; // dichiarazione anticipata della classe DepositSlot
9
10   class Deposit : public Transaction
11   {
12   public:
13      Deposit( int, Screen &, BankDatabase &, Keypad &, DepositSlot & );
14      virtual void execute(); // esegue la transazione
15   private:
16      double amount; // importo da depositare
17      Keypad &keypad; // riferimento al tastierino dell'ATM
18      DepositSlot &depositSlot; // riferimento allo sportello dei depositi
19      double promptForDepositAmount() const; // ottiene l'importo da depositare
20   }; // fine della classe Deposit
21
22   #endif // DEPOSIT_H
```

Figura E.21 Definizione della classe Deposit

```
1    // Deposit.cpp
2    // Definizione delle funzioni membro della classe Deposit.
3    #include "Deposit.h" // definizione della classe Deposit
```

Figura E.22 Definizione delle funzioni membro della classe Deposit (continua)

```
4   #include "Screen.h" // definizione della classe Screen
5   #include "BankDatabase.h" // definizione della classe BankDatabase
6   #include "Keypad.h" // definizione della classe Keypad
7   #include "DepositSlot.h" // definizione della classe DepositSlot
8
9   const static int CANCELED = 0; // costante che rappresenta l'annullamento
10
11  // il costruttore inizializza i dati membro
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  {
18     // corpo vuoto
19  } // fine del costruttore della classe Deposit
20
21  // esegue la transazione; riscrive la funzione virtuale pura della classe base
22  void Deposit::execute()
23  {
24     BankDatabase &bankDatabase = getBankDatabase(); // ottiene il riferimento
25     Screen &screen = getScreen(); // ottiene il riferimento
26
27     amount = promptForDepositAmount(); // ottiene l'importo dall'utente
28
29     // verifica se l'utente ha inserito il valore di un importo
30     if ( amount != CANCELED )
31     {
32        // chiede di depositare la busta nell'apposito sportello
33        screen.displayMessage(
34           "\nPlease insert a deposit envelope containing " );
35        screen.displayDollarAmount( amount );
36        screen.displayMessageLine( " in the deposit slot." );
37
38        // riceve la busta del deposito
39        bool envelopeReceived = depositSlot.isEnvelopeReceived();
40
41        // verifica se è stata inserita una busta
42        if ( envelopeReceived )
43        {
44           screen.displayMessageLine( "\nYour envelope has been received."
45              "\nNOTE: The money just will not be available until we"
46              "\nverify the amount of any enclosed cash, and any enclosed "
47              "checks clear." );
48
49           // aggiorna il conto corrente dell'utente
50           bankDatabase.credit( getAccountNumber(), amount );
51        } // fine dell'if
52        else // busta non ricevuta
```

Figura E.22 Definizione delle funzioni membro della classe Deposit (continua)

```
53          {
54            screen.displayMessageLine( "\nYou did not insert an "
55               "envelope, so the ATM has canceled your transaction." );
56          } // fine dell'else
57       } // fine dell'if
58       else // operazione cancellata
59       {
60          screen.displayMessageLine( "\nCanceling transaction..." );
61       } // fine dell'else
62    } // fine della funzione execute
63
64    // chiede all'utente di inserire un importo in centesimi
65    double Deposit::promptForDepositAmount() const
66    {
67       Screen &screen = getScreen(); // riferimento allo schermo
68
69       // visualizza la richiesta e ottiene la risposta
70       screen.displayMessage( "\nPlease enter a deposit amount in "
71          "CENTS (or 0 to cancel): " );
72       int input = keypad.getInput(); // riceve l'importo digitato
73
74       // verifica se è stato specificato un importo corretto
75       if ( input == CANCELED )
76          return CANCELED;
77       else
78       {
79          return static_cast< double >( input ) / 100; // resituisce l'importo
                                                           in dollari
80       } // fine dell'else
81    } // fine della funzione promptForDepositAmount
```

Figura E.22 Definizione delle funzioni membro della classe Deposit

Definizione delle funzioni membro della classe Deposit

La figura E.22 presenta l'implementazione della classe `Deposit`. La riga 3 include la definizione della classe mentre le righe 4-7 includono le definizioni delle altre classi utilizzate dalle funzioni membro della classe. La riga 9 dichiara la costante `CANCELED` che corrisponde al valore che l'utente inserisce quando vuole annullare l'operazione.

Come la classe `Withdrawal` anche la classe `Deposit` possiede un costruttore (righe 12-19) che passa tre parametri al costruttore della classe base utilizzando gli inizializzatori della classe base (riga 15). Il costruttore prende inoltre i parametri `atmKeypad` e `atmDepositSlot` che assegna ai corrispondenti dati membro (riga 16).

La funzione membro `execute` (righe 22-62) riscrive la funzione virtuale pura `execute` della classe base `Transaction` con un'implementazione concreta che esegue i passi necessari a portare a termine una transazione di tipo deposito. Le righe 24-25 ottengono i riferimenti al database della banca ed allo schermo della macchina ATM. La riga 27 chiede all'utente di inserire l'importo del deposito invocando la funzione d'utilità privata `promptForDepositAmount` (definita alle righe 65-81) e memorizza il valore restituito nel dato membro `amount`. La funzione membro `promptForDepositAmount` chiede all'utente

di inserire l'importo da depositare espresso in centesimi (perchè il tastierino della macchina ATM non contiene il punto decimale e questo è vero per molte macchine ATM reali) e restituisce il valore `double` in dollari corrispondente.

La riga 67 nella funzione membro `promptForDepositAmount` ottiene un riferimento allo schermo della macchina ATM e le righe 70-71 visualizzano un messaggio sullo schermo che chiede all'utente di inserire l'importo del deposito in centesimi o il valore "0" per annullare l'operazione. La riga 72 riceve il dato dell'utente inserito dal tastierino. L'istruzione `if` alle righe 75-80 determina se l'utente ha inserito un valore per l'importo da depositare o ha scelto di annullare l'operazione. Nel secondo caso la riga 76 restituisce il valore `CANCELED` altrimenti la riga 79 restituisce l'importo convertito in dollari eseguendo un cast della variabile `input` al tipo `double` e dividendo la variabile per `100`. Se l'utente inserisce il valore `125`, ad esempio, la riga 79 restituisce il valore `125.0` diviso per `100` ovvero `1.25` (125 centesimi di dollaro equivalgono a $1.25).

L'istruzione `if` alle righe 30-61 nella funzione membro `execute` determina se l'utente ha scelto di annullare l'operazione. In tal caso la riga 60 visualizza un apposito messaggio e l'esecuzione della funzione termina. Se invece l'utente ha inserito un importo le righe 33-36 chiedono all'utente di inserire una busta di deposito con l'importo di valuta specificato. Ricordate che la funzione membro `displayDollarAmount` della classe `Screen` visualizza un valore di tipo `double` formattato come valuta.

La riga 39 imposta una variabile booleana locale al valore restituito dalla funzione membro `isEnvelopeReceived` dell'oggetto `depositSlot` che indica se la busta del deposito è stata ricevuta. Ricordate che abbiamo implementato la funzione membro `isEnvelopeReceived` (righe 7-10 di figura E.10) in modo che restituisca sempre il valore `true` perché stiamo simulando il funzionamento dello sportello dei depositi e assumiamo che l'utente abbia inserito la busta. Nell'implementazione della funzione membro `execute` della classe `Deposit`, tuttavia, viene considerata la possibilità che l'utente non abbia inserito la busta perché una buona ingegneria del software richiede che vengano presi in considerazione tutti i possibili valori restituiti da una funzione. La classe `Deposit` è così preparata a future versioni della funzione `isEnvelopeReceived` che restituiscano anche il valore `false`. Le righe 44-50 vengono eseguite solo se la busta è stata ricevuta: le righe 44-47 visualizzano un appropriato messaggio all'utente e la riga 50 accredita l'importo del deposito sul conto dell'utente nel database. Le righe 54-55 vengono invece eseguite se la busta non viene inserita: viene visualizzato un messaggio che informa l'utente che la transazione è stata annullata e la funzione termina l'esecuzione senza modificare il conto dell'utente.

E.13 Il programma di verifica ATMCaseStudy.cpp

Il file `ATMCaseStudy.cpp` (figura E.23) è un semplice programma C++ che ci consente di "mettere in funzione" la macchina ATM e verificare il funzionamento della nostra implementazione. La funzione `main` (righe 6-11) non fa altro che istanziare un oggetto `ATM` chiamato semplicemente `atm` (riga 8) e invocare la sua funzione membro `run` (riga 9) per iniziare l'applicazione.

```
1    // ATMCaseStudy.cpp
2    // Programma di verifica del progetto ATM.
3    #include "ATM.h" // definizione della classe ATM
4
```

Figura E.23 Il file ATMCaseStudy.cpp che inizia l'applicazione ATM (continua)

```
5    // la funzione main crea ed esegue il sistema ATM
6    int main()
7    {
8       ATM atm; // crea un oggetto ATM
9       atm.run(); // fa partire il sistema
10      return 0;
11   } // fine del main
```

Figura E.23 Il file ATMCaseStudy.cpp che inizia l'applicazione ATM

E.14 Conclusioni

Congratulazioni! Avete completato il progetto orientato agli oggetti della macchina ATM. Ci auguriamo che abbiate trovato questa esperienza utile e che abbia contribuito a chiarire i concetti che avete imparato nel libro. Qualsiasi vostro commento, critica o suggerimento è ben accetto. Potete rivolgervi all'indirizzo `deitel@deitel.com`: vi risponderemo prontamente.