1 **Java: The ATM App**

CST242

2 **The Analysis Stage**

- The analysis state focuses on defining the problem to be solved which requires two things
  - *Solve the problem right* (correctly)
  - *Solve the right problem*
- It is the job of system analysts to collect the requirements that indicate the specific problem to solve

3 **Requirements Documents (Page 1)**

- A requirements document is a document that contains all the requirements to a certain system
- Typically written from a user's point-of-view by the user/client to allow people to understand what the system should do

4 **Requirements Documents (Page 2)**

- Usually it is the result of detailed requirements gathering which might include interviews with:
  - Possible users of the system
  - Specialists in fields related to the system

5 **Requirements Documents (Page 3)**

- The requirements document generally should avoid anticipating or defining "how" the system will do its job
  - This allows interface designers and engineers to later use their expertise to provide the optimal solution to the requirements

6 **Requirement Document for ATM (Page 1)**

- A bank plans to install a new ATM to allow customers to perform financial transactions
- Each user has one bank account
- ATM users should be able to view their account balance, withdraw cash and deposit funds

7 **Requirement Document for ATM (Page 2)**

- The user interface of the ATM contains:
  - A screen that displays messages to the user
  - A keypad that receives numeric input from the user (only *digits* and an <Enter> key)
  - A cash dispenser that dispenses cash to the user
  - A deposit slot that receives deposit envelopes from the user

8 **Requirement Document for ATM (Page 3)**

- The cash dispenser starts each day loaded with 500 $20 bills
- An ATM session consists of authenticating a user based on an account number and

personal identification number (PIN)

10 **Requirement Document for ATM (Page 4)**

- ATM transaction software is being developed at now—the hardware "will be developed later"
- This version should use the computer's screen to simulate the ATM screen and the computer's keyboard to simulate the keypad:
  - The ATM asks the user to type the account number on the keypad rather than reading it from a bank card
  - All output including receipts (not printed) appears on the screen
-

11 **Requirement Document for ATM (Page 5)**

- To authenticate a user and perform transactions, ATM must interact with the bank's information database of accounts
- For each account in the database is stored:
  - An account number
  - A PIN
  - The account balance

12 **Requirement Document for ATM (Page 6)**

- Upon approaching the ATM, the user should experience the following:
  1. The screen displays Welcome! and prompts the user to enter an account number
  2. The user enters a five-digit account number using the keypad
  3. The screen prompts the user to enter the PIN that is associated with that account number
  4. The user enters a five-digit PIN using the keypad

13 **Requirement Document for ATM (Page 7)**

- Upon approaching the ATM, the user should experience the following (*con*.):
  5. If the user is authenticated, the screen displays the main menu; otherwise the screen displays an error message and the ATM returns to *Step 1*

15 **Requirement Document for ATM (Page 8)**

- The main menu should contain a numbered option for each of the three transaction types:
  - Balance inquiry (option 1)
  - Withdrawal (option 2)
  - Deposit (option 3)
- It also should also contain an option to exit the system (option 4)
- The user selects an option by typing a number onto the keypad

16 **Requirement Document for ATM (Page 9)**

- If the user enters 1 to make a balance inquiry, the screen displays the user's account balance

– The ATM must retrieve the account balance (both total balance and available balance) from the bank's database to do so

17 **Requirement Document for ATM (Page 10)**

- If the user enters 2 to make a withdrawal:
  1. The screen displays a menu containing standard withdrawal amounts:
     - $20 (option 1)
     - $40 (option 2)
     - $60 (option 3)
     - $100 (option 4)
     - $200 (option 5)

     The menu also contains an option to cancel the transaction (option 6)

19 **Requirement Document for ATM (Page 11)**

- If the user enters 2 to make a withdrawal (*con*.):
  2. The user enters a menu selection using the keypad
  3. If the withdrawal amount is greater than the account balance (available balance), the screen displays that message, tells the user to select a smaller amount and the ATM returns to *step 1*

     If the withdrawal amount is less than or equal to the account balance, the ATM proceeds to *step 4*

     If the user chooses to cancel the transaction, the ATM displays the main menu and waits for input

20 **Requirement Document for ATM (Page 12)**

- If the user enters 2 to make a withdrawal (*con*.):
  4. If the cash dispenser contains enough cash, the ATM proceeds to *step 5*

     Otherwise the screen displays a message indicating the problem, prompts the user to select a smaller amount and then returns to *step 1*
  5. The ATM debits (subtracts) the withdrawal amount from the user's account
  6. The cash dispenser "dispenses" amount to the user
  7. The screen displays a message reminding the user to take the money

21 **Requirement Document for ATM (Page 13)**

- If the user enters 3 to make a deposit:
  1. The screen prompts the user to enter a deposit amount or 0 (zero) to cancel
  2. The user enters a deposit amount or 0 (zero) using the keypad

     Note: The keypad has no decimal point or dollar sign so the user only can type digits; the ATM then divides the input by 100 to get dollars and cents (for example 1234 ÷ 100 = 12.34)

22 **Requirement Document for ATM (Page 14)**

- If the user enters 3 to make a deposit (*con*.):
  3. If the user specifies a deposit amount, the ATM proceeds to *step 4*

     If the user chooses to cancel the transaction, the ATM displays the main menu and

waits for input

4. The screen displays a message telling the user to insert a deposit envelope

23 **Requirement Document for ATM (Page 15)**

- If the user enters 3 to make a deposit (*con*.):
  5. If the deposit slot receives the envelope within two minutes, ATM credits user's account (money not immediately available: verify cash in envelope or checks must clear—credit available balance once this occurs)
     If deposit slot does not receive the envelope, screen prints that message
     Either way the ATM then displays the main menu and waits for user input

24 **Requirement Document for ATM (Page 16)**

- After the ATM executes a transaction, it should return to the main menu so that the user can perform additional transactions
- If the user exits the system, the screen should display a "Thank You Message" and then display the "Welcome Message" for the next user

25 **Use Case Modeling (Page 1)**

- Use case modeling involves identifying the use cases of the system
- Each use case represents a specific capability (action or event) that the system provides to its clients (actors)
- The goal is to show the kinds of interactions users will have with the system without providing details

26 **Use Case Modeling (Page 2)**

- In an ATM system the use cases might be:
  - "View Account Balance"
  - "Withdraw Cash"
  - "Deposit Funds"
  - "Transfer Funds between Accounts"
  - "Make a payment"
- This ATM implements the first three

27 **Use Case Diagrams (Page 1)**

- A use case diagram (or case diagram) is a *simple* form of UML diagram which is a representation of a user's interaction with the system
- Created during the analysis (the first) stage of the software life cycle to identify the different types of users of a system and the different use cases
  - Shows relationship between user and different use cases
- Often accompanied by informal text that gives more detail like the text in requirements documents

28 **Use Case Diagrams (Page 2)**

- It has been said before that "Use case diagrams are the blueprints for your system"– McLaughlin , B., Pollice, G., & West, D (2006)
- Use case diagrams convey the intent of the system in a more *simplified* manner to

stakeholders
  – Stakeholders are the individuals, group or organization who may affect or be affected by the system
- They are "interpreted more completely than class diagrams"—Siau, K. & Lee, L. (2004)

29 **Use Case Diagrams (Page 3)**

- The use case diagram is the first of 13 diagram types that are used for documenting system models according to the UML 2 standard
- Other diagrams are:
  – Class diagrams
  – State machine diagrams
  – Activity diagrams
  – Communication diagrams (or collaboration diagrams)
  – Sequence diagrams

32 **Identifying the Classes in a System (Page 1)**

- It is possible to identify the classes in a system by looking at the *nouns* and *noun phrases* in the requirements document, e.g.:

bank	money/funds		account number	ATM
screen	PIN	user	keypad	
bank database		customer	cash dispenser	balance inquiry
transaction		$20 bill/cash	withdrawalaccount	
deposit slip		deposit	balance	deposit envelope

33 **Identifying the Classes in a System (Page 2)**

- The following can be eliminated:
  – "bank" as it is not part of the ATM
  – "customer" and "user" since they merely interact with the ATM and are not part of it
  – "$20 bill/cash" is not actually being automated but can be managed and represented by the cash dispenser
  – "deposit envelope" since out system does not say what happens to them after they are deposited; simply acknowledging that the deposit slot receives them is satisfactory

34 **Identifying the Classes in a System (Page 3)**

- The following can be eliminated (*con.*):
  – "balance" and "account number" and "PIN" represent attributes rather than behaviors
  – "transaction" is more a generalized notion of three specific behaviors, "balance inquiry" and "withdrawal" and "deposit"
    - The object-oriented notion of inheritance will bring "transaction" back later for its common elements with the three specific behaviors

35 **Identifying the Classes in a System (Page 4)**

- This leaves the remaining probable system classes:

- ATM
- Screen
- Keypad
- Cash Dispenser
- Deposit Slot
- Account
- Bank Database
- Balance Inquiry
- Withdrawal
- Deposit

36 **Class Diagrams (Page 1)**

- UML class diagrams allow us to model classes in a system along with their interrelationships
- Each class is modeled as a rectangle with three compartments:
  - Top contains the class name centered
  - The middle contains the class attributes
  - The bottom contains the class operations (behaviors)
- Initially the middle and bottom compartments may be left blank since attributes and operations are "yet to be determined"

38 **Class Diagrams (Page 2)**

- Class diagrams also show the relationship between classes of the system
- Solid lines connect two classes and represent what is called an association

40 **Navigability**

- Adding association lines to a UML class diagram show which objects need references to other objects
- Navigability arrows (→) added to association lines show the direction in which an association between two classes is traversed
- Sometimes relationship goes in a single direction and sometimes it is *bidirectional*

42 **Visibility (Access Modifiers)**

- private—for instance variables so that they cannot be seen output the object; indicated by minus sign (-) in UML diagram
- public—for methods that are accessed from other classes; indicated by plus sign (+) in UML diagram
- NOTE: Methods called "utility methods" that only are accessed by and serve methods of the *same class* should be private

43 **Identifying Class Attributes (Page 1)**

- Class attributes are implemented as fields (instance variables)
- Attributes may be identified in the requirements document as descriptive words and phrases
- For each significant word and phrase, create an attribute and assign it to a class that

may need it

44 **Identifying Class Attributes (Page 2)**

- Classes and possible attributes for the ATM:

Class	Descriptive words and phrases
ATM	user is authenticated
Screen	(*none*)
Keypad	(*none*)
Cash Dispenser	each day begins with 500 $20 bills
Deposit Slot	(*none*)
Account	account number, PIN, balance
Bank Database	(*none*)
Balance Inquiry	account number
Withdrawal	account number, amount
Deposit	account number, amount

46 **Identifying Class Operations (Page 1)**

- Class operations are implemented as methods
- An operation is a "service" that an object of the class provides to a client (user) of the class
- Attributes may be identified in the requirements document as descriptive verb and verb phrases
- Then relate these verbs and phrases to classes in the system

47 **Identifying Class Operations (Page 2)**

- Classes and possible operations for the ATM:

Class	Descriptive verbs and phrases
ATM	executes financial transactions
Screen	displays a message to user
Keypad	received numeric input from user
Cash Dispenser	dispenses cash, confirms sufficient cash for withdrawal
Deposit Slot	receives a deposit envelope
Account	receives an account balance, credits deposit amount to account, debits withdrawal amount from account

49 **Identifying Class Operations (Page 3)**

- Classes and possible operations for the ATM (*con*.)

Class	Descriptive verbs and phrases
Bank Database	authenticates a user, receives an account balance, credits deposit amount to account, debits withdrawal amount from account

Balance Inquiry (*none*)
Withdrawal (*none*)
Deposit (*none*)

51 **Implement the ATM System**

1. Convert the classes into code which represent the "skeleton" of the application
2. Modify the code to include inheritance
3. Complete the code for the ATM system

52 **Try It Out**

- Add the "skeleton" classes (header and empty body):
  - ATM, BankDatabase, Account, Keypad, Screen, CashDispenser, DepositSlot
- Create class Withdrawal
  - Add the class including constructor (with no code as a "stub" placeholder)
  - Add instance variable attributes:
    - *accountNumber*, *amount*
  - Add instance variables for associations to ATM objects:
    - *screen*, *bankDatabase*, *keypad*, *cashdispenser*
  - Add the execute() method "stub"

53 **Adding Inheritance**

- Find commonality among classes
- Create inheritance hierarchy to implement this commonality in a more efficient and elegant way
- The common elements are placed in a superclass
- The superclass may be abstract and may contain abstract methods
  - Abstract element names are designated in *italics* in the UML class diagram

57 **Try It Out**

- Add super abstract class Transaction
  - Move instance variable *accountNumber* and instance variables for associations *screen* and *bankDatabase* from class Withdrawal to Transaction
  - Add the constructor "stub"
  - Add public abstract void execute();

58 **Try It Out**

- Add the two other "Transaction" subclasses:
  class Deposit extends Transaction
  class BalanceInquiry extends Transaction
  - Add constructor with parameters and call to superclass method super()
  - Add execute() method with @Override annotation to both classes

59 **Try It Out**

- Continue sub class Withdrawal
  - Add extends Transaction to class signature

– Complete the constructor with parameters and call to superclass method super()
– Add @Override annotation to execute() method

60 **Complete the App**

- Following specifications in the UML diagrams, complete the rest of the application

61 **Class Screen**

- Represents the screen of the ATM as output to the console
- The three methods in this class print:
  System.out.print()—with no carriage return
  System.out.println()—with a carriage return
  System.out.printf()—formats type double variables where output format is like String.format()

63 **Class Keypad**

- Represents the keypad of the ATM by declaring a Scanner object as an instance variable and instantiating it in the constructor method
- The two methods in this class get input by calling:
  nextInt()—inputs an int from the Scanner
  nextDouble()—inputs a double from the Scanner

65 **Class ATM**

- Starts ATM running by calling the run() method which loops continually until operator "breaks" into the system
- This instance variable is initialized to false but modified to true when the user is authenticated
  private boolean userAuthenticated;
- Account number instance variable for the "current user"
  private int currentAccountNumber;

66 **Class ATM—Simulated Objects**

- There are five instance variables which are references to associated ATM simulated objects
  private final Screen screen;
  private final Keypad keypad;
  private final CashDispenser cashDispenser;
  private final DepositSlot depositSlot;
  private final BankDatabase bankDatabase;

67 **Class ATM—Constants**

- There are four constants which correspond to the "Main Menu" options
  private final static int BALANCE_INQUIRY = 1;
  private final static int WITHDRAWAL = 2;
  private final static int DEPOSIT = 3;
  private final static int EXIT = 4;

69 **Class ATM—Constructor**

- The no-parameter constructor initializes instance variables to reflect that a user is not yet logged in

```
userAuthenticated = false;
currentAccountNumber = 0;
```

- … and instantiates the five ATM simulated objects

```
screen = new Screen();
keypad = new Keypad();
cashDispenser = new CashDispenser();
depositSlot = new DepositSlot();
bankDatabase = new BankDatabase();
```

70 **Class ATM—run() Method**

- The outer while loop continues indefinitely

```
while (true)
```

- The inner while loop authenticates user

```
while (! userAuthenticated)
```

- Call to perform the transactions for current user

```
performTransactions();
```

- Reset authentication and account number so another user may login

```
userAuthenticated = false;
currentAccountNumber = 0;
```

71 **Class ATM—authenticateUser() Method**

- Displays prompts to the Screen and gets user input from the Keypad for *account number* and *PIN*
- Updates boolean instance variable *userAuthenticated* from BankDatabase method authenticateUser()

```
userAuthenticated = bankDatabase.authenticateUser( accountNumber, pin );
```

- Updates *accountNumber* instance variable or sends error message to Screen based upon user authentication

```
if (userAuthenticated)
{ …
```

72 **Class ATM—performTransactions() Method**

- Declares and attempts to instantiate a Transaction object from the constructor of one of its subclasses, and if successful call its execute() method
- Calls "helper" method createTransaction() to instantiate object if displayMainMenu() returns BALANCE_INQUIRY, WITHDRAWAL or DEPOSIT
- Displays an error message to the Screen if user enters an invalid option, and repeats loop
- The loop continues for currently logged in user until displayMainMenu() returns EXIT

73 **Class ATM—displayMainMenu() Method**

- Displays the "Main Menu" to the Screen and gets input from the Keypad as the user choice
- Returns the user choice as an int

74 **Class ATM—createTransaction() Method**

- Declares, instantiates (by calling one of the three subclass constructors) and return's a Transaction object

```
transaction = new BalanceInquiry( accountNumber, screen, bankDatabase );
transaction = new Withdrawal( accountNumber, screen, bankDatabase, keypad,
    cashDispenser );
transaction = new Deposit( accountNumber, screen, bankDatabase,  keypad,
    depositSlot );
```

- Based upon value of int type parameter which is either BALANCE_INQUIRY, WITHDRAWAL or DEPOSIT

75 **Try It Out**

- Start class ATM
  - Instantiate ATM object and call run() from main()
  - Except for method run(), all other methods are "helper methods" and should be private
  - To complete run(), first complete class Screen
  - To *start* method authenticateUser(), first complete class Keypad
  - Complete all of authenticateUser() *except* call to bankDatabase.authenticateUser()

76 **Try It Out**

- Start class ATM (*con*.)
  - Create just the "stubs" and complete *later*:
    - performTransactions()
    - displayMainMenu()
    - createTransaction()

77 **Class Account**

- Each object represents one bank account
- Account number for the account

```
private int accountNumber;
```

- PIN number related to account number for the account

```
private int pin;
```

- Funds available for withdrawal

```
private double availableBalance;
```

- Funds available plus pending deposits

```
private double totalBalance;
```

78 **Class Account—Constructor**

- Initializes the *accountNumber*, *pin*, *availableBalance* and *totalBalance* from the parameters

80 **Class Account—validatePin() Method**

- Determines whether the user input PIN parameter matches the PIN in this account object

```
return (this.pin == pin);
```

81 **Class Account—the get Methods**

- Returns the *accountNumber*, *availableBalance* and *totalBalance* instance variables
  - The *pin* number is "secret" and as such never returned

82 **Class Account—credit() Method**

- Credits the account by adding deposit *amount* parameter to the *totalBalance* instance variable

```
totalBalance += amount;
```

  - The *amount* is not added to the *availableBalance* until "... we verify the amount of any enclosed cash and your checks clear" (simulated)

83 **Class Account—debit() Method**

- Debits the account by subtracting withdrawal *amount* parameter from the *availableBalance* and *totalBalance* instance variables

```
availableBalance -= amount;
totalBalance -= amount;
```

84 **Class BankDatabase**

- Stores and manages an array of Account objects
- Class BankDatabase is a "driver" for the Account class in that accounts are instantiated and their public methods called from a BankDatabase object
- The instance variable is the array of accounts

```
private Account[] accounts;
```

85 **Class BankDatabase—Constructor**

- Instantiates the array to a specified number of elements

```
accounts = new Account[3];
```

- Instantiates the Account elements and assigns values to *accountNumber*, *pin*, *availableBalance* and *totalBalance*

```
accounts[0] = new Account(12345, 54321, 1000.00, 1200.00);
...
```

87 **Class BankDatabase—getAccount() Method (Page 1)**

- This is a private "helper" method which serves the other methods in this BankDatabase class:

```
getAvailableBalance()
getTotalBalance()
credit()
debit()
```

88 **Class BankDatabase—getAccount() Method (Page 2)**

- Loops through the account objects in the *accounts* array
  for (Account account : accounts)
- If the account number of one of those *account* elements matches the *accountNumber* parameter …
  if (account.getAccountNumber() == accountNumber)
- … returns that *account*
  return account;
- If no match and for each loop completes, returns null
  return null;

89 **Class BankDatabase—authenticateUser() Method**

- Attempts to instantiate an Account object by passing the *accountNumber* parameter to "helper" method getAccount() of this class BankDatabase
  Account account = getAccount(accountNumber);
- If the account is instantiated, returns boolean value from the method validatePin() of class Account
  if (account != null)
  return account.validatePin(pin);
- If the account fails to be instantiated …
  return false;

90 **Class BankDatabase—getAvailableBalance() Method**

- Gets an Account object by passing the *accountNumber* parameter to "helper" method getAccount() of this class BankDatabase
  getAccount(accountNumber)
- Then returns int from getAvailableBalance() method from the Account class
  return getAccount(accountNumber) .getAvailableBalance();

91 **Class BankDatabase—getTotalBalance() Method**

- Gets an Account object by passing the *accountNumber* parameter to "helper" method getAccount() of this class BankDatabase
  getAccount(accountNumber)
- Then returns int from getTotalBalance() method from the Account class
  return getAccount(accountNumber) .getTotalBalance();

92 **Class BankDatabase—credit() Method**

- Gets an Account object by passing the *accountNumber* parameter to "helper" method getAccount() of this class BankDatabase
  getAccount(accountNumber)
- Then credits the account by passing the *amount* parameter to method credit() from the Account class
  getAccount(accountNumber) .credit(amount);

93 **Class BankDatabase—debit() Method**

- Gets an Account object by passing the *accountNumber* parameter to "helper" method getAccount() of this class BankDatabase
  getAccount(accountNumber)
- Then debits the account by passing the *amount* parameter to method debit() from the Account class
  getAccount(accountNumber) .debit(amount);

94 **Class Transaction**

- Subclasses of the abstract class Transaction are BalanceInquiry, Withdrawal and Deposit
- An account number for this transaction
  private final int accountNumber;
- An ATM Screen object for output for this transaction
  private final Screen screen;
- An ATM BankDatabase object to access information about the Account for this transaction
  private final BankDatabase bankDatabase;

95 **Class Transaction—Constructor**

- Initializes the *accountNumber*, *screen* and *bankDatabase* instance variables from the parameters
- These variables are accessed by the three subclasses through the *get* methods
- Class Transaction is "immutable" as it has no *set* methods

96 **Class Transaction—the get Methods**

- Method getAccountNumber() is called by the three subclasses to return the *accountNumber* for a transaction
- Method getScreen() is called by subclasses as a "helper" method to instantiate a *screen* object for transaction output
- Method getBankDatabase() is called by subclasses as a "helper" method to instantiate a *bankDatabase* object for accessing account information for the transaction

97 **Try It Out**

- Complete class Transaction
- Complete the constructor header and call to super for:
  class BalanceInquiry
  class Withdrawal
  class Deposit

98 **Try It Out**

- Complete class ATM
  – Complete method authenticateUser()
  – Complete methods displayMainMenu() and createTransaction() before the method

performTransactions()

99 **Class BalanceInquiry**

- Class BalanceInquiry extends Transaction and represents an ATM balance inquiry transaction

101 **Class BalanceInquiry—Constructor**

- Passes the parameters *accountNumber*, *screen* and *bankDatabase* to superclass constructor and instantiates objects for AccountNumber, Screen and BankDatabase

102 **Class BalanceInquiry—execute() Method (Page 1)**

- Overrides the abstract execute() method of class Transaction and performs the balance inquiry transaction
- Instantiates *bankDatabase* and *screen* objects by calling superclass Transaction "helper" methods

```
BankDatabase bankDatabase = getBankDatabase();
Screen screen = getScreen();
```

103 **Class BalanceInquiry—execute() Method (Page 2)**

- Gets the *availableBalance* and *totalBalance* from the BankDatabase object
  – Method getAccountNumber() is inherited from superclass Transaction

```
double availableBalance = bankDatabase.getAvailableBalance( getAccountNumber() );
double totalBalance = bankDatabase.getTotalBalance( getAccountNumber() );
```

104 **Class BalanceInquiry—execute() Method (Page 3)**

- Displays the *availableBalance* and *totalBalance* to the Screen object

```
...
screen.displayDollarAmount( availableBalance );
...
screen.displayDollarAmount( totalBalance );
...
```

105 **Class DepositSlot**

- Class DepositSlot simulates whether or not the envelope is received
- Its single method (there is no constructor) always returns true since this is just a simulation of a real deposit slot

```
public boolean isEnvelopeReceived()
{
   return true;
}
```

107 **Class Deposit**

- Class Deposit extends Transaction and represents an ATM deposit transaction
- The instance variable *amount* is the amount of the deposit
- The associated ATM simulated objects:
  – *keypad* is used to select the amount of withdrawal from the "Withdrawal Menu"

– *depositSlot* is used to simulate the slot in the ATM for depositing cash

- The constant corresponds to a value of zero (0) which is the "canceled" option

```
private static final int CANCELED = 0;
```

109 **Class Deposit—Constructor**

- Passes the parameters *accountNumber*, *screen* and *bankDatabase* to superclass constructor and instantiates objects for AccountNumber, Screen and BankDatabase
- The parameters *keypad* and *depositSlot* are assigned to the instance variables to instantiate the Keypad and DepositSlot objects

110 **Class Deposit—execute() (Page 1)**

- Overrides the abstract execute() method of class Transaction and performs the deposit transaction
- Instantiates *bankDatabase* and *screen* objects by calling superclass Transaction "helper" methods

```
BankDatabase bankDatabase = getBankDatabase();
Screen screen = getScreen();
```

- Calls method promptForDepositAmount() to get the *amount* of the deposit

```
amount = promptForDepositAmount();
```

111 **Class Deposit—execute() (Page 2)**

- First checks to see if the user selected *CANCELED* option

```
if (amount == CANCELED)
```

- Is so, reports "Canceling…"
- Is not, "receives envelope" to *depositSlot*

```
boolean envelopeReceived = depositSlot.isEnvelopeReceived();
```

112 **Class Deposit—execute() (Page 3)**

- If envelope is received (which always will be true in this simulation) …

```
if (envelopeReceived)
{ …
```

- Finally credits the account into the *bankDatabase* object

– Method getAccountNumber() is inherited from superclass Transaction

```
bankDatabase.credit( getAccountNumber(), amount);
```

113 **Class Deposit—promptForDepositAmount()**

- Instantiates a Screen object from method getScreen() of the Transaction class to display the prompt

```
Screen screen = getScreen();
```

- Displays prompt for the deposit amount to the Screen object and gets *input* from the Keypad object
- For *input* values of zero (0) or less, returns *CANCELED*, e.g. zero (0)

```
return CANCELED;
```

- For all positive values, returns the *input* value

```
return input;
```

114 **Class CashDispenser**

- Simulates the cash dispenser of the ATM
- Constant stores the initial number of $20 bills that are stored in the cash dispenser
  private final static int INITIAL_COUNT = 500;
- Instance variable keeps track of how much cash (how many $20 bills) currently are available
  private int count;

115 **Class CashDispenser—Constructor**

- The constructor sets *count* instance variable (how many $20 bills) to the value of constant INITIAL_COUNT
  count = INITIAL_COUNT;

117 **Class CashDispenser—dispenseCash() Method**

- Calculates the number of $20 bills from the *amount* parameter divided by 20
  amount / 20
- Assign number of bills required to local variable
  int billsRequired = amount / 20;
- Subtract *billsRequired* from the *count* instance variable
  count -= billsRequired;

118 **Class CashDispenser—isSufficientCashAvailable() Method**

- Calculates the number of $20 bills from the *amount* parameter and assigns result to local variable
  int billsRequired = amount / 20;
- "Calculates" a boolean value that indicates whether or not the count of $20 bills is sufficient
  return (count >= billsRequired);

119 **Class Withdrawal (Page 1)**

- Class BalanceInquiry extends Transaction and represents an ATM withdrawal transaction
- The instance variable *amount* is the amount of the withdrawal
- The associated ATM simulated objects:
  – *keypad* is used to select the amount of withdrawal from the "Withdrawal Menu"
  – *cashDispenser* is used to simulate the dispensing of cash for the withdrawal

120 **Class Withdrawal (Page 2)**

- The constants represent the corresponding options for the "Withdrawal Menu"

```
private static final int DOLLARS_20 = 1;
private static final int DOLLARS_40 = 2;
private static final int DOLLARS_60 = 3;
private static final int DOLLARS_100 = 4;
private static final int DOLLARS_200 = 5;
```

```
private static final int CANCEL = 6;
```

122 **Class Withdrawal—Constructor**

- Passes the parameters *accountNumber*, *screen* and *bankDatabase* to superclass constructor and instantiates objects for AccountNumber, Screen and BankDatabase
- The parameters *keypad* and *cashDispenser* are assigned to the instance variables to instantiate the Keypad and CashDispenser objects

123 **Class Withdrawal—execute() (Page 1)**

- Overrides the abstract execute() method of class Transaction and performs the withdrawal transaction
- The boolean instance variable *transactionComplete* controls the loop and remains false until either cash is dispensed or the user cancels the operation
- The double instance variable *availableBalance* gets the amount available from the *bankDatabase* for this account

124 **Class Withdrawal—execute() (Page 2)**

- Instantiates *bankDatabase* and *screen* objects by calling superclass Transaction "helper" methods

```
BankDatabase bankDatabase = getBankDatabase();
Screen screen = getScreen();
```

125 **Class Withdrawal—execute() (Page 3)**

- The do while loop continues until true is assigned to the variable *completeTransaction* because either:
  - An *amount* to withdraw is selected from menu
  - The user selects *CANCEL* from menu

```
do
   amount = displayMenuOfAmounts();
   ...
while (! completeTransaction);
```

  - Method displayMenuOfAmounts() gets the *amount* of withdrawal from the user

126 **Class Withdrawal—execute() (Page 4)**

- First checks to see if the user selected the *CANCEL* option

```
if (amount == CANCEL)
```

- Is so, reports "Canceling..." and ends the loop

```
transactionComplete = true;
```

- Is not, continue by getting the *availableBalance* for the account from the *bankDatabase*

```
availableBalance = bankDatabase.getAvailableBalance( getAccountNumber() );
```

127 **Class Withdrawal—execute() (Page 5)**

- Is there *not* a sufficient available balance in the account for the withdrawal

```
if (amount > availableBalance)
```

- Is there *not* sufficient cash in the ATM for the withdrawal
  if (! cashDispenser. isSufficientCashAvailable(amount))
- These errors are reported to the user and the loop repeats to display the "Withdrawal Menu" again

128 **Class Withdrawal—execute() (Page 6)**

- If all is good, debits the account from the *bankDatabase* object, removes that *amount* of cash from *cashDispenser* object and stops the loop
  – Method getAccountNumber() inherited from superclass Transaction
  bankDatabase.debit(getAccountNumber(), amount);
  cashDispenser.dispenseCash(amount);
  transactionComplete = true;

129 **Class Withdrawal—displayMenuOfAmounts() (Page 1)**

- Instantiates a Screen object from method getScreen() of the Transaction class to display the menu
  Screen screen = getScreen();
- The int variable *userChoice* controls the loop and is assigned the value zero (0)
  int userChoice = 0;
- An int[] array stores the *amounts* related to the values in the "Withdrawal Menu"
  int[] amounts = {0, 20, 40, 60, 100, 200};

130 **Class Withdrawal—displayMenuOfAmounts() (Page 2)**

- The variable *userChoice* controls the loop, the menu is displayed to the Screen object, and the user's choice is input from the Keypad object

```
while (userChoice == 0)
{
   screen.displayMessageLine( "\nWithdrawal Menu:" );
   screen.displayMessageLine("1 - $20");
   ...
   int input = keypad.getInt();
   ...
```

131 **Class Withdrawal—displayMenuOfAmounts() (Page 3)**

- For all dollar amounts from the "Withdrawal Menu" the value is assign from the int[] *amounts* array (which also ends the loop)
  userChoice = amounts[input - 1];
  – Menu *input* goes from 1 to 5, array indexes from 0 to 4
- If "Cancel Transaction" is selected from the "Withdrawal Menu" the loop is terminated
  userChoice = CANCEL;
- For invalid *input* values, an error is reported to the user, the value of *userChoice* remains zero (0) and the loop repeats