Chapter 1 - Object Oriented Software Design

Objective

After reading this chapter you:

- Understand the concept of Object Oriented Design
- Will be able to apply Object Oriented Design principles to designing software systems.
- Will understand UML class relationships – dependency and association – in order to decompose problem definitions into component.
- Will understand the concepts – cohesion and coupling – in order to assemble the components (classes) into complete programs.

Introduction

Object-oriented design (OOD) is the philosophy of developing an object-oriented model of a software system, by defining the classes and their interactions with one another. The major benefits of OOD are easy maintainable systems, easy understandable systems, easy expandable systems, and reusable components of a system.

In this chapter you will learn the principles of OOD. You will learn about the different kinds of relationships that can be formed among class diagrams, how to use these relationships to model the solution to an entire system. OOD demands that the programmer defines principles to decompose problem definitions into separate entities; it also demands that the programmer determines the relationship among the entities, which eventually leads to the solution of the problem. The principle that we will use to achieve the former is called cohesion; and the principle to achieve the latter is called coupling.

The chapter closes with a survey of misinterpretations and pitfalls that could occur during the design and implementation phases. Before studying the pitfalls, however, we will develop an entire system using the concept of Object Oriented Design along with the Uniform Modeling Language.

UML Class Relationship Diagram

The Unified Modeling Language (**UML**) as we know from volume I is a standardized specification language that uses a set of diagrams to model objects, with the ultimate aim of solving problems. UML features several types of diagrams for different purposes - component diagrams, composite structure diagram, deployment diagram, object diagram, package diagram, and class diagram. We will continue to use class diagram, this time using it to solve more complex problems

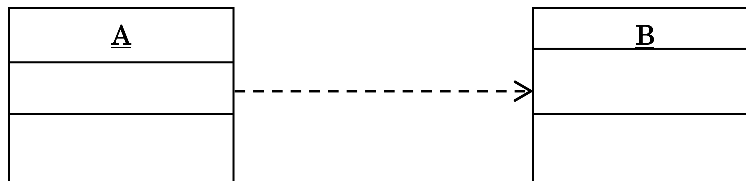
than we had encountered in volume I. As we have mentioned then, one of the applications of the **UML** is to design class diagrams.

The term UML class diagram really means UML Class Relationship diagram. There are four types of class relationship diagrams that can be formulated using UML notation. They are UML Dependency diagram, UML Association diagram, UML Generalization diagram, and UML Realization diagram. We will discuss class dependency and association diagrams in this chapter. Generalization and realization diagrams will be discussed in chapter 2 when we discuss inheritance.

Dependency Diagram

The UML dependency relationship is the least formal of the four relationship diagrams, as we will see shortly. The default of any relationship is bi-directional. However, when applied to designing class diagrams it is limited to unidirectional relationship. The UML dependency diagram uses a broken line to show the relationship between two classes. In addition, an open arrow head is used to show the directional relationship between classes. **Figure 1.1** shows two classes **A** and **B** that have unidirectional dependency. The relationship between these two classes means that class **A** depends on class **B**.

Figure 1.1 UML Class diagram; class A depends on class B



In **Figure 1.1**, the unidirectional dependency class diagram means that class **A** uses class **B**. In this situation, the unidirectional dependency relationship is restricted to the following meaning.

(a) Class **A** receives an instance of class **B** as a parameter to at least one of its methods, or
(b) Class **A** creates an instance of class **B**, local to one of its methods.

In this kind of dependency relationship, class **B** cannot be an attribute of class **A**. Hence class **A** cannot contain an instance of **B**.

Listing 1.1 shows the interpretation of the UML unidirectional dependency relationship. Notice that method1 in class **A**, accepts reference parameter **b**. Also, method2 in class **A** creates a reference of **B** local to method2 in **A**.

Listing 1.1 UML dependency relationship between class A and class B

```
1.  public class A
2.  {
3.      public void method1(B b)
4.      {
5.      }
6.      public void method2()
7.      {
8.          B b = new B();
9.      }
10. }
```

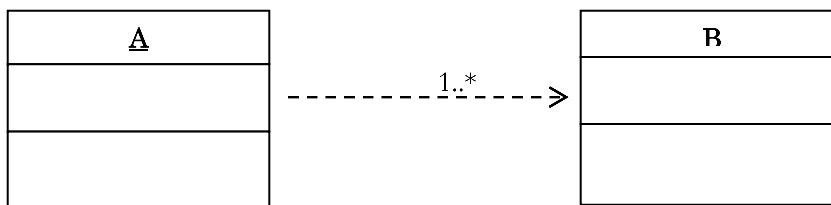
```
1.  public class B
2.  {
3.      public void method()
4.      {
5.      }
6.      public void method2()
7.      {

8.      }
9.  }
```

It is not enough to know that one class depends on another, but equally important is to know the frequency on which it depends the other. The following table is a frequency chart showing the possible frequency occurrences.

Frequency	Meaning
0.. 1	Zero or one time
1	Only once
0 .. *	Zero or more times
1 .. *	1 or more times
n	Only n times, n > 1
0 .. n	Zero or more times, where n > 1
1 .. n	1 or more times, where n > 1

Figure 1.2 shows another feature of a UML class dependency diagram. Not only does the class A depends on the class B, but it depends on it from one to any number of times, as indicated by the symbols above the arrow.

Figure 1.2 UML Class diagram; class A may depend on class B multiple times



Self-Check

1. Given that A and B are two classes, and that class B depends on class A. Draw a unidirectional dependency relationship diagram between both classes.

2. Given that A and B are two classes. What must be true, in order to establish a unidirectional dependency between class B and class A.

3. Given that Q represents a class. Which of the following classes establish a dependency relationship of class P on class Q?

```
(a) class P
  {
      Q q;
      P()
      {
          q = new Q();
      }
  }
```

```
(b) class P
    {
        P()
        {
          Q q = new Q();
        }
    }
```

```
(c) class P
    {
        P()
        {

        }
        void add( Q q);
        {

        }
    }
```

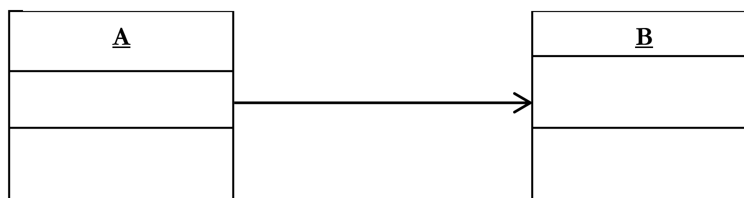
```
(d) class P
    {
        P()
        {

        }
    }
```

```
(e) class P
    {
        P()
        {
        Q q = new Q();
        }
    }
```

Association Diagram

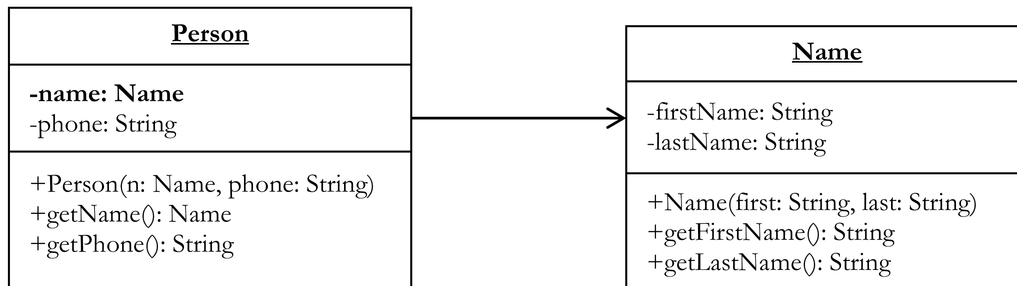
A class association diagram defines a relationship that is much stronger than dependency relationship. **Figure 1.3** shows the UML association relationship class diagram between class **A** and class **B**. The solid line with an open ended arrow establishes a unidirectional association relationship between these classes. The strength of an association relationship class diagram means that class **A** will contain at least one instance variable of class **B**, which makes class **B** structurally a part of class **A**.

Figure 1.3 A UML Association relationship diagram



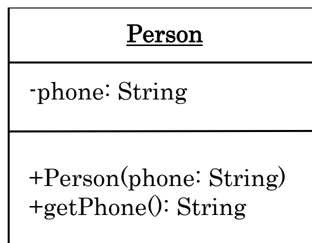
Example 1.1 Let us consider the classes **Person** and **Name** as shown in **Figure 1.4**. The class **Person** is a composition of two fields, the first which is of type **Name**, the second of type **String**. This relationship establishes the fact that the class Name forms part of the physical structure of the class Person.

Figure 1.4 Association diagram of class Person with respect to class Name.



If we were to remove the field **name** from the class Person, then the class Person, having just the field, **phone**, would not establish that we are talking about person. Hence by including the field, **name** to Person, at least intuitively adds the meaning person. **Figure 1.5** shows the class Person without the field **name**.

Figure 1.5 class Person without the field name



Self-Check

1. What condition(s) must exist for the class A to have an association relationship on class B?

2. If class A has an association relationship with class B, draw the association diagram between both classes.

3. Given that Q represents a class. Which of the following classes establish an association relationship of class P on class Q?

```
(a) class P
    {
        Q q;
        P()
        {
            q = new Q();
        }
    }
```

```
(b) class P
    {
        P()
        {
            Q q = new Q();
        }
    }
```

```
(c) class P
    {
        P()
        {

        }
        void add( Q q);
        {

        }
    }
```

```
(d) class P
    {
        P()
        {

        }
    }
```

```
(e) class P
    {
        P()
        {
            Q q = new Q();
        }
    }
```

4. There are two classes, Circle and Shape. Define both classes where the class Shape defines an association relationship on the class Circle.

5. Define two classes, Sentence and Words, where the class Sentence has an association relationship with an array of potential Word objects.

6. Using Question 5, draw a unidirectional association relationship between both classes.

Cohesion

In object oriented programming design, the solution to a problem may be so complex that it may involve several classes. The entire set of classes in the design is sometimes referred to as the software system; and each class in the system is referred to as a component of the system. The design and implementation of large-scale software systems draw attention to the need for well-defined design methodologies and modeling techniques that can reduce the complexity of the solution, and at the same time increase the probability of a correct solution. While there may not be a one-shop, quick-fix solution to good software design, there are some well proven methodologies that have been used as guidelines towards good software designs. Two of methodologies are cohesion and coupling. In this section we will discuss cohesion, and the next section we will discuss coupling.

With respect to object oriented design, the concept of cohesion focuses on the logical construction of each component within the system; where each component is required to concentrate on a single objective. In turn, each module, or method within the component should be responsible to carry out one precise task.

The quality of a software system is generally measured by the strength, or the cohesiveness of each of the components. The strength of a cohesive system is measured from low cohesive to high cohesive. A low cohesive system is a system that focuses on more than tasks. The more tasks it has to perform, is the weaker the cohesiveness of the system. A highly cohesive system on the other hand focuses on only one task. A lowly cohesive system is considered to be a poorly designed system; whereas, a highly cohesive system is considered to be of a good design.

A highly cohesive method has several advantages than a very low cohesive one. A highly cohesive method is simpler to understand, simpler to re-use, easier to maintain, easier to code, and is easier to debug. If methods are highly cohesive, then the class itself will be highly cohesive. Hence the class is easy to understand, because it is designated to relay a single complete thought. Above all, a change in one component may not necessitate a change in any of the other components. In an environment where there is low cohesion, errors are harder to detect and correct. In an effort to correct an error, you may inadvertently create new ones.

As we have said, the concept of cohesion can be readily applied to object oriented software design, in that the design requires the programmer to decompose problem definitions into highly cohesive components. Rarely are all systems purely cohesive. Some components may have to establish relationship such as dependency relation, or association relation among other components.

Example 1.2 Consider the following narrative:

Design a Java program that carries out banking operations on customers' account. The concept of banking is characterized by customers and their account balances. The program should be able to:

- Store the customers' account information in a database.
- Make deposits.
- Make withdrawals.
- Search for an account, given the account number.
- Delete an account from the active accounts in the database, and store any deleted account into a separate database

Solution I

A programmer who does not know about the concept of cohesion would more likely write a single class, along with a test class, to solve problems of this kind. This class would encapsulate all of the characteristics described in the problem. That is, the class would be responsible for:

- The collection of data and dissemination of information for names.
- The collection data and dissemination of information for addresses.
- The collection of data and dissemination of information for customer.
- Create bank accounts, update bank account, and dissemination of information about bank accounts.
- Create database, store bank accounts in database, search database for accounts, and remove accounts from database.

A system of this kind would be considered loosely cohesive; one that would be difficult to debug if there is a logic error; and almost impossible to maintain if any segment requires change to it.

Solution II

A second approach would be to combine the concept of name, address, customer, and bank account as one component; thus keeping the database component separate. But just like the first solution, the bank account component would have too much responsibility. Let us consider a third possibility.

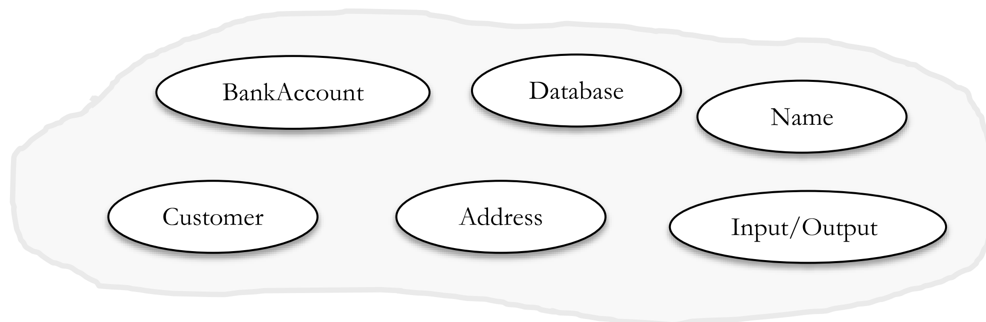
Solution III

When we analyze the problem, we see that in order to have a highly cohesive system, there are at least six components, excluding the test class. These components are as follow:

- A **bank account** component that is characterized by customer and an initial balance.
- With respect to customer, a **customer** is an entity that is characterized by name, address, and account number.
- When it comes to address, an **address** is characterized by street, city, state, and zip code.
- A name can also be characterized as a component which has at least a first name and a last name.
- The concept database, which is a repository for banking transactions, can also be considered another component.
- Lastly, we may need a component that can be used to read data and display information.

Figure 1.6 shows the system consisting of these six components, not including the test class. At the moment there is no defined relationship between any of them. It is only a decomposition of the various components within the system.

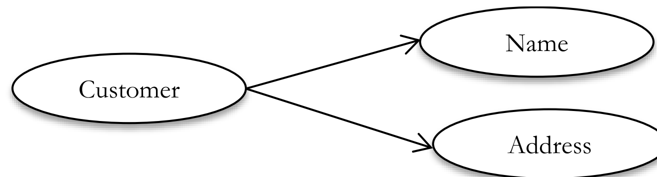
Figure 1.6 A software system of six components



In general, when designing an object oriented software system, the first thing that you want to do is to determine what the possible components are, and what the responsibility of each will be. In the current example, the entity Name will define names only; the entity Address will be restricted to facilitating address only. As it stands, there is no relationship between a name and an address object. Should there be an error within any of the two, then we would directly focus on the one that has the problem. There would be no need to interfere with the other entity. These two components are now said to be highly cohesive.

In terms of customer on the other hand, in real life a customer has a name and an address. Against this background, the entity Customer will have both attributes - Name and Address. That is, both entities must be physically a part of the Customer entity. This consideration therefore establishes an association relationship between the component Customer, and the components Name and Address. This situation is represented by **Figure 1.7**.

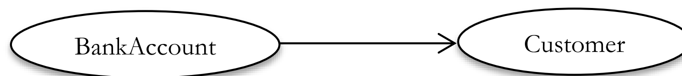
Figure 1.7 Association relationship between Customer, and the pair Name **and** Address



With regards to the ease of detecting and correcting errors, if we know for sure that the entities Name and Address are clear of any logic errors, but the component Customer has some form of logic error, then the problem must lie with the current component. Either that it has introduced new variables that are causing the problem; or, the established components are being used improperly.

With regards to the component Bank account, a bank account obviously has customer as one of its attributes. In this situation the component BankAccount has to establish an association relationship between itself and the component, Customer, as shown in **Figure 1.8**.

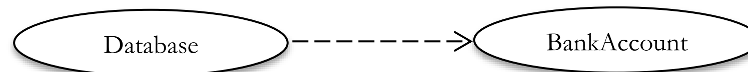
Figure 1.8 Association relationship between BankAccount and Customer



In terms of maintenance or debugging purposes, if we know for sure that the component Customer is flawless, and a problem surfaces in the BankAccount entity, then without any question we would narrow our effort to the local definition of the component, in terms of its newly defined variables and methods.

With regards to the entity Database, it receives bank account objects, via one of its methods, and store them. This establishes a dependency relation of Database upon BankAccount. See **Figure 1.9**.

Figure 1.9 Dependency relationship of Database upon BankAccount

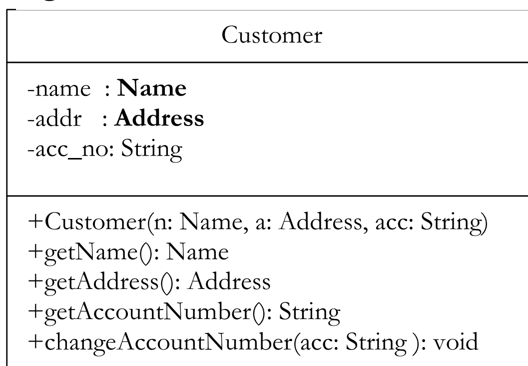


Any component such as BankAccount appearing as field in the Database would be secondary to the operation of storing and retrieving data. In other words, whether or not such fields are included, they would not greatly influence the definition of the entity Database. This runs parallel to the discussion in **Example 1.1**, where we discussed dependency relationship.

The next step in the design process is to construct the class notation diagram for each of these components. We will use the names in **Figure 1.6**, for the name in each of the class notation diagram that is to be drawn. Against this background, it does not matter the order in which the diagrams are drawn; what matters is the relationship one entity will have on another one. In this regard, let us design the UML diagram for the class Customer.

The narrative tells us what attributes constitute a customer object. That is, with respect to customer, a customer is characterized by name, address, and account number. When we analyze this statement, it is obvious that the fields for the component Customer are: component type Name, component type Address, and a String type, for the account number. This leads us to conclude that the fields Name and Address will form an association relationship with the class Customer. See **Figure 1.10**. Although not specified in the problem, it is possible that a customer may want to change the account number of any number of reasons. Against this background we include a mutator method that does exactly that.

Figure 1.10 Fields Name and Address define association relationship with Customer



As you will notice, this component is solely responsible for addressing customers' information; it does not address the concerns of any of the other components. From all indications, each of the methods, by nature of their names and return type, will focus one task; thus making the component itself highly cohesive.

As you would have noticed in Figure 1.10, no mention was made about the physical structure of the classes Name and Address; yet it is possible to use them to code the class Customer, as shown in **Listing 1.2**.

Listing 1.2 Class Customer

```
1.  public class Customer
2.  {
3.      private Name name;
4.      private String acc_no;
5.      private Address address;
6.
7.      public Customer (Name n, String ac, Address addr)
8.      {
9.          name = n;
```

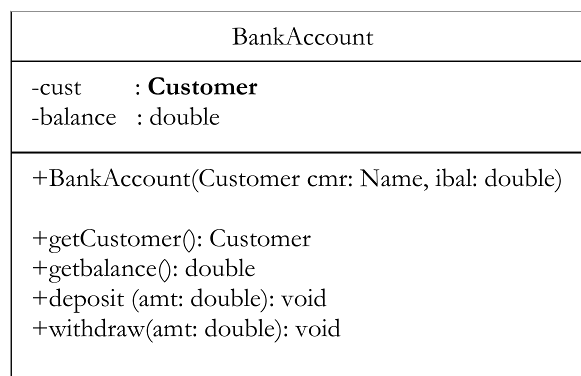
```
10.         acc_no = ac;
11.         this.address = addr;
12.     }
13.     public Name getName() {
14.         return name;
15.     }
16.     public Address getAddress() {
17.         return address;
18.     }
19.     public String getAccountNumber() {
20.         return acc_no;
21.     }
22.     void changeAccountNumbsr(String acc) {
23.         acc_no = acc;
24.      }
25. }
```

Next, we will design the class BankAccount. This component we know is characterized by Customer objects, and an initial balance, as stated in the opening sentence of the problem description. This means that there is an association relation between itself and the class Customer.

It is customary that when an account is opened, an initial deposit is also made. This tells us that the constructor will not only accept Customer object, but also an initial deposit which will be used to offset the balance in the account upon creating the account. This implies that it is necessary to have a field that will act as an accumulator for the balance.

The problem also specifies that one should be able to make deposits and withdrawal any number of times. These activities necessitate a mutator method for each – deposit and withdraw. In addition, we will also need accessor method for each of the fields. **Figure 1.11** shows the class notation diagram representing the entity BankAccount.

Figure 1.11 The entity BankAccount



Whereas the method withdraw seems straightforward; i.e., a withdrawal is a subtraction, the account could end up with a negative balance. To avoid this from happening, one approach is to test

whether or not the amount to be withdrawn exceeds the actual balance in the account; and if this is the case, we may want to let the program alert the customer. If we take this approach, then we will see that the method **withdraw** will have three responsibilities – testing the data, alerting customer by providing a message, and withdrawing, by subtracting values. In this context this method is lowly cohesive, since it has three responsibilities. This in turn weakens the cohesiveness of the class itself. To strengthen the cohesiveness of this method would be to let it carry out the subtraction for withdrawal only; then we would design a separate method to do the testing. In addition, let the class that is implementing this aspect of this component determine the alert message. In this context, this version has a higher degree of cohesiveness than the previous one. In particular, see the method called isSufficient. **Figure 1.12** shows a modified version of the component BankAccount.

Figure 1.12 The entity BankAccount

BankAccount
-cust : **Customer** -balance : double
+BankAccount(Customer cmr: Name, ibal: double) +getCustomer(): Customer +getbalance(): double +deposit (amt: double): void +withdraw(amt: double): void +isSufficient(amt: double): boolean

In the design of this system, the entities Name and Address are quite simple, when compared to the other components. The entity Name is comprised of three fields of type String – first name, last name and middle name (if any). Usually an entire name is not changed; but may be a last name may change due to marriage. In a case like this we include a mutator method for that purpose. Since this component has nothing else than addressing the possible concerns of a name, it is considered a highly cohesive component. See **Figure 1.13**.

Listing 1.3 shows the definition of the class BankAccount.

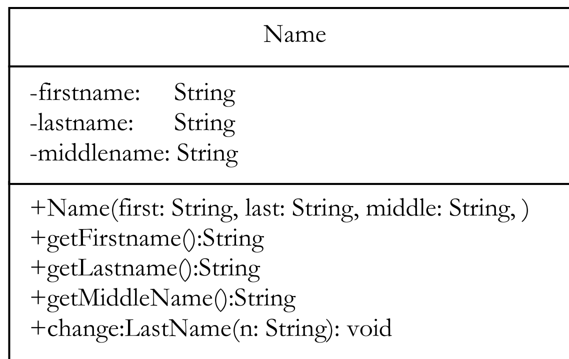
Listing 1.3 Class BankAccount

```
1.  public class BankAccount
2.  {
3.      private double balance;
4.      private Customer cust;
5.
6.      public BankAccount (Customer c, double amt)
7.      {
8.          cust = c;
```

```
9.              balance = amt;
10.         }
11.         public Customer getCustomer() {
12.             return cust;
13.         }
14.         public void deposit(double amt) {
15.             balance += amt;
16.         }
17.         public void withdraw(double amt) {
18.             balance -= amt;
19.         }
20.         public double getAmount() {
21.             return balance;
22.         }
23.         public boolean isSufficient(double amt) {
24.             return balance >= amt;
25.         }
26.  }
```

Figure 1.13 shows the UML class notation diagram representing the name object of an individual. Notice that we have included a mutator method that can be used to change ones last name, if there is ever the need to do so.

Figure 1.13 The entity Name



Listing 1.4 shows the definition of the class Name.

Listing 1.4 Definition of the class Name

```
1.   class Name
2.   {
3.       String first;
4.       String last;
5.
6.       Name(String f, String l)
7.       {
```

```
8.              first = f;
9.              last = l;
10.         }
11.
12.         String getFirst(){
13.              return first;
14.         }
15.
16.         String getLast(){
17.              return last;
18.         }
19.  }
```

Like the component Name, the component Address is highly cohesive. It is comprised of five fields of type String – street, city, zip, and state, country. This component also is highly cohesive. See **Figure 1.14**.

Figure 1.14 The entity Address

Address
-street: String -city: String -state: String -zip: String -country: String
+Address(street: String, city: String, state: String, zip: String) +getStreet() : String +getCity() : String +getState( ) : String +getZip() : String +getCountry(): String

Listing 1.5 shows the definition of the class Address.

Listing 1.5 The definition of the class Address

```
1.    class Address
2.    {
3.          String street, city, state,  zip;
4.
5.          Address(String str, String city, String st, String zip)
6.          {
7.                street = str;
8.                this.city = city;
9.                state = st;
10.               this.zip = zip;
11.         }
```

```
12.         String getStreet() {
13.             return street;
14.         }
15.         String getCity() {
16.             return city;
17.         }
18.         String getState() {
19.             return state;
20.         }
21.         String getZip() {
22.             return zip;
23.         }
24.  }
```

Analyzing the entity Database we see that it is more complex than all of the other components. But as complex as this may seem however, its singly purpose as far as the banking activities are concern is a focus on database operations only:

- Add bank accounts to a growing list[1] of bank accounts.
- Searches the list for a given account.
- Obtain a copy of a bank account if it is in the database.
- Find the location of a bank account, if that account is in the database.
- Remove an account from the list of accounts.

On the surface, **Figure 1.15** seems to be an accurate response to the five requirements above. The method, **add**, accepts a bank account object and adds it to the list; the method **delete**, removes the account from the list; the method **search**, searches the list and returns its index; and the method **getAccount**, simply returns an account from the list.

Figure 1.15 The entity Database

Database
-list : ArrayList -account : BankAccount
+ Database () +add(account: BankAccount): void +delete(accountNumber: String): BankAccount +search (accountNumber: String): int +getAccount(accountNumber: String): BankAccount

On closer examination of this solution, there are several assumptions that were made. The method add undoubtedly has only one function; that is, it appends an account to the growing list of accounts.

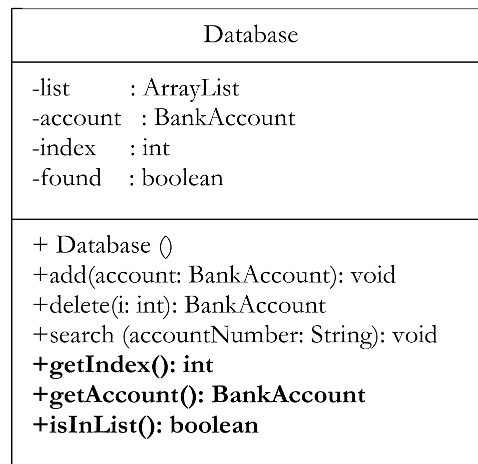
[1] For an expandable list, the class ArrayList is more appropriate than an array which is non-expandable.

The method delete has the potential of doing more than one tasks. The acceptance of the account number as parameter, suggests that it uses the account number to conduct a search for the bank account. Not only does it search the list, but it also removes the account object from the list if it is there, but it also returns a copy of it. This method exhibits low cohesiveness. A better approach is for the method to receive the index, and use it to remove and return the object that it removes.

The method search should be a mutator method, designed to provide three pertinent pieces of information that are consistent with a search - whether or not an item is in the list; if the item is in the list, where in the list it is; and thirdly, which account it is, if it is in the list. With this modification there should now be three accessor methods, one for each of the three pieces of information produced by the search method. This modification strengthens the degree of cohesiveness of both the method; hence the class on the whole is strengthened.

Lastly, the method getAccount, by accepting as parameter the account number, has the potential of executing multiple tasks – searching, determining if the account is in the list, and returning a copy of that object. This method should depend on the outcome of the search method, and should only be called if an account is found to be in the list. That is, the method search should be made to accept an account number as parameter, and conduct the search as discussed in the previous paragraph. **Figure 1.16** shows a more cohesive class than the one shown in Figure 1.15.

Figure 1.16 The entity Database



Listing 1.6 shows the class definition of the UML diagram representing the entity, Database.

Listing 1. 6

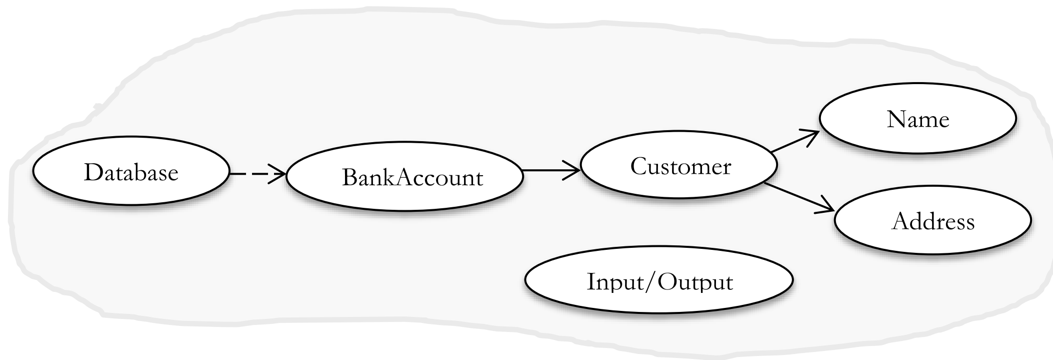
```
1.  import java.util.ArrayList;
2.  class Database {
3.          ArrayList<BankAccount> list;
4.          BankAccount ba;
```

```
5.          int index;
6.          boolean found;
7.          Database(){
8.                 list = new ArrayList<BankAccount>();
9.          }
10.         void search(String key) {
11.                found = false;
12.                int i = 0;
13.
14.                while(!found && i < list.size()) {
15.                       BankAccount b = list.get(i);
16.                       if(b.getCustomer().getAccountNumber().equalsIgnoreCase(key))
17.                       {
18.                              ba = b;
19.                              found = true;
20.                              index =i;
21.                       }
22.                       else
23.                               i++;
24.                }
25.         }
26.         void add(BankAccount b) {
27.                 list.add(b);
28.          }
29.         BankAccount delete(int i) {
30.                return list.remove(i);
31.         }
32.         int getIndex(){
33.                return index;
34.         }
35.         boolean inList(){
36.                return found;
37.         }
38.         BankAccount getAccount() {
39.                return ba;
40.         }
41.         int size()  {
42.             return list.size();
43.             }
44.         boolean isEmpty()  {
45.                return list.isEmpty();
46.         }
47.         ArrayList getList(){
48.                 return list;
49.         }
50.  }
```

As we have stated, the quality of a software system is generally measured by the cohesiveness of each of the components. If the components when connected do not have cycles of dependencies, but form a perfect tree, then the system is in general a highly cohesive. The result of the above

analysis, with the exception of the component called Input/Output, results in a tree, as shown in **Figure 1.17**. A highly cohesive system generally gives rise to a lowly coupled system.

Figure 1.17 The graph of a cohesive system generally forms a perfect tree



Self-Check

1. Which of the following statements is true concerning a class? Select one.
   (a) In a class, fields can ONLY be user-defined types.
   (b) In a class, fields can ONLY be primitive data types or existing Java types (from existing Java classes)
   (c) In a class, fields can be primitive data types, existing Java types, or user-defined types.
   (d) In a class, fields can ONLY be primitive data types such as int, float, etc.

2. Write a class called Person. A person has a name and a social number. Assume that a class Name exists with first name and last name. Design the class Person such that it accepts a Name object and a social security number. Provide accessor methods to return the Name object and also the social security number.

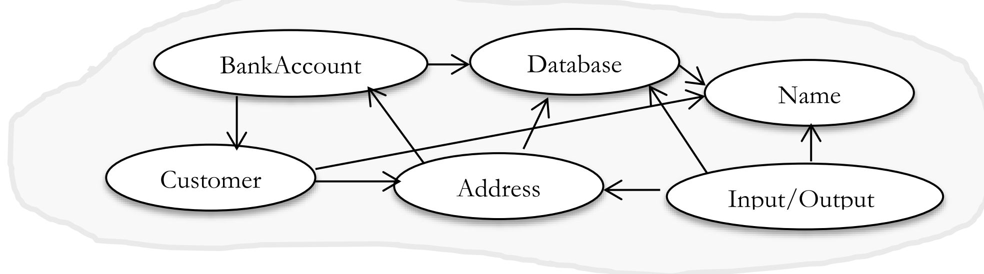
   Write a class called Registration that accepts two values - a Person object, and an array called courses [ ] - representing strings of course titles. No student is allowed to register for more than five (5) at any one time. Write a method that determines if a student has registered for too many courses. You are responsible for providing the requisite variables, and any other methods deemed necessary.

Coupling

Whereas cohesion describes the relationship of elements within a single component of a system, coupling on the other hand describes the interdependent relationships among the components within the system. In order for the system to work harmoniously, the components themselves must be connected in such a way that they work harmoniously with one another as well. The nature of the connections is important, as it will determine the stability of the system. For instance, if there is a change in one component of the system, will change require change in any other component; and if so, to what extent.

In objected oriented design, if every component has a reference to every other component in a system, then this interdependent relationship is said to be tightly coupled. In a tightly coupled system, often we find cyclic relationship among components. **Figure 1.18** shows a highly coupled system. In the figure the components BankAccount, Customer and Address form a cycle.

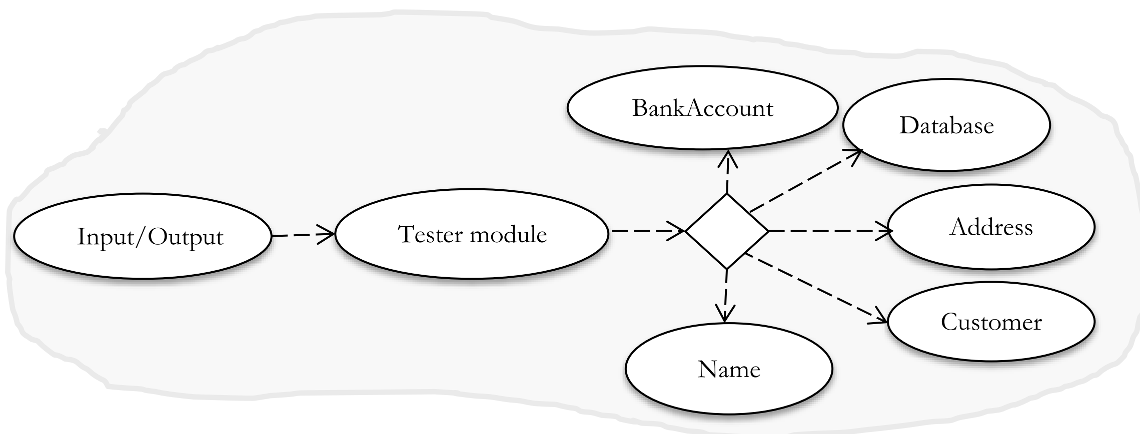
Figure 1.18 A highly coupled system



Tightly coupled systems generally present major programming challenges. A tightly coupled system increases the possibility of too much information flow through it, which in term has the potential of creating redundancies along the way. In a tightly coupled system, a change in one component could have ripple effect of changes in other components. As a result, modification to a tightly coupled system can prove difficult, if not impossible to maintain.

Instead of creating a system that is highly coupled, it is better to create loosely coupled one, by designing a separate component that supervises the cohesive portion of the system. In general, a loosely coupled system is more desirable than a tightly coupled one. **Figure 1.19** shows a loosely coupled system of components that is supervised by a tester component.

Figure 1.19 A loosely coupled system



This system represents one of dependency relationship, rather than an association relationship. Notice that there is no cyclic relationship among any of the components. In addition, the system is governed by a decision control node that accepts input generated from the Input/Ouput component, and selects the component that is to be accessed. As the arrows show, no one component is directly

in control of another component. In its implementation, the tester component will provide options for selecting which component gets accessed. This solution is best illustrated by a flowchart. See **Figure 1.20**.

Figure 1.20 Test component coordinates the other components

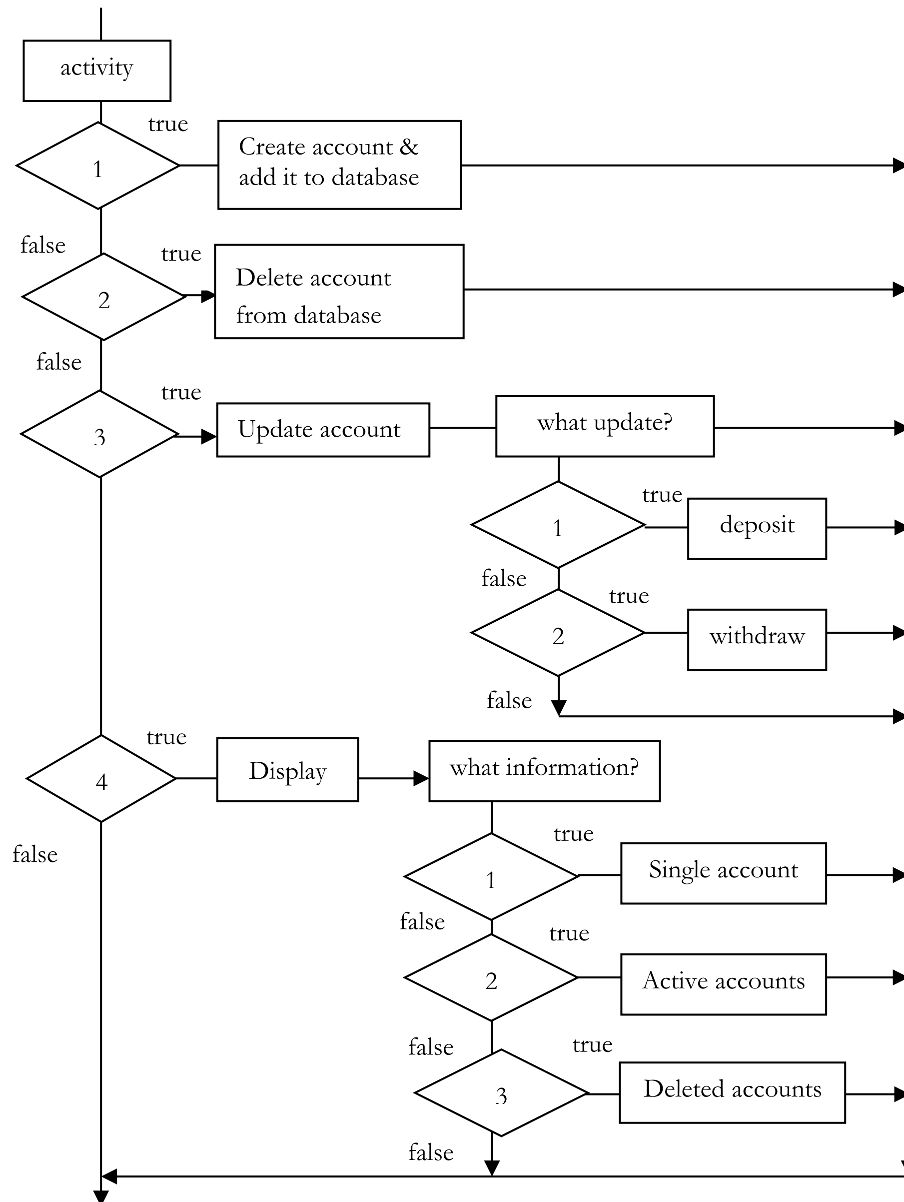


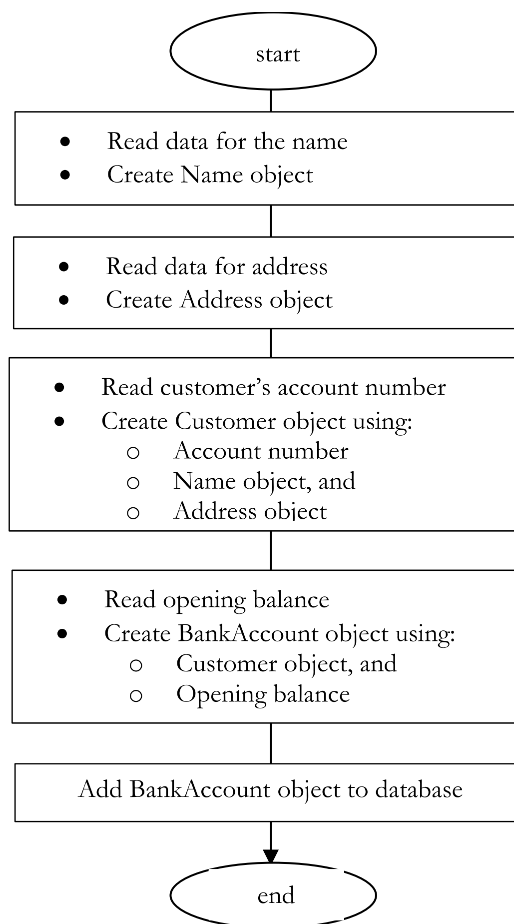
Figure 1.20 illustrates four major activities – create a new account and add it to the database, delete an existing account from the database, update an existing account, and display information.

Prior to the creation of the various components object for a BankAccount we first create a database object for keeping a list of the current customers, and one for the list of closed account. Perhaps we could create these objects as follows:

```
Database db = new Database( );
Database close = new Database( );
```

The creation of a BankAccount object and adding it to the database is best described by the flowchart shown in **Figure 1.21.** In this situation the user inputs the data needed to create the respective object, in the order shown in the flowchart.

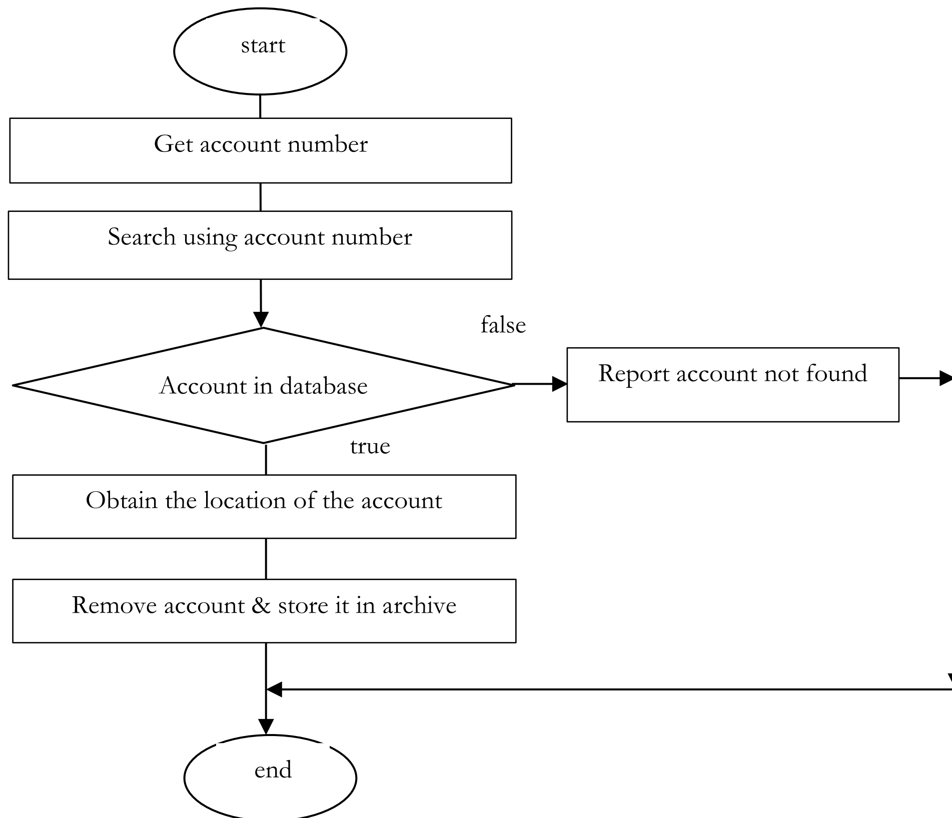
Figure 1.21 Creating a BankAccount object



The deletion of an account, and the storing of it, first require determining whether or not the account exists. To determine if the account exists you will need the account number in order to conduct the search. Once it is determined that the account exists, then it is just a matter of obtaing the location where it it resides, and remove it, and store it among the list of deleted accounts. If the

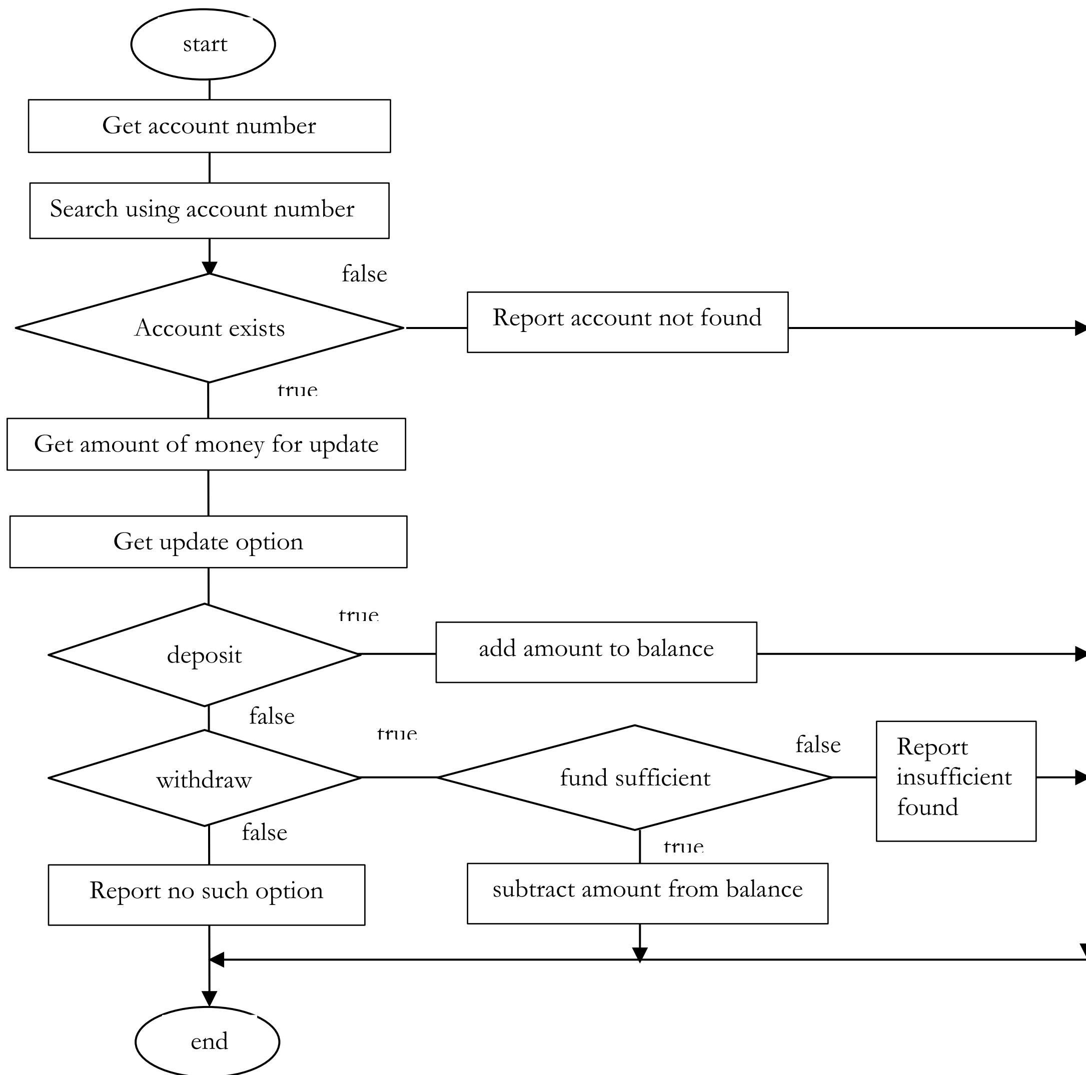
accoount does not exists, inform the user that it does not exists. This algorithm is illustrated in **Figure 1.22**.

Figure 1.22 Deleting an account and storing it in a list of deleted accounts



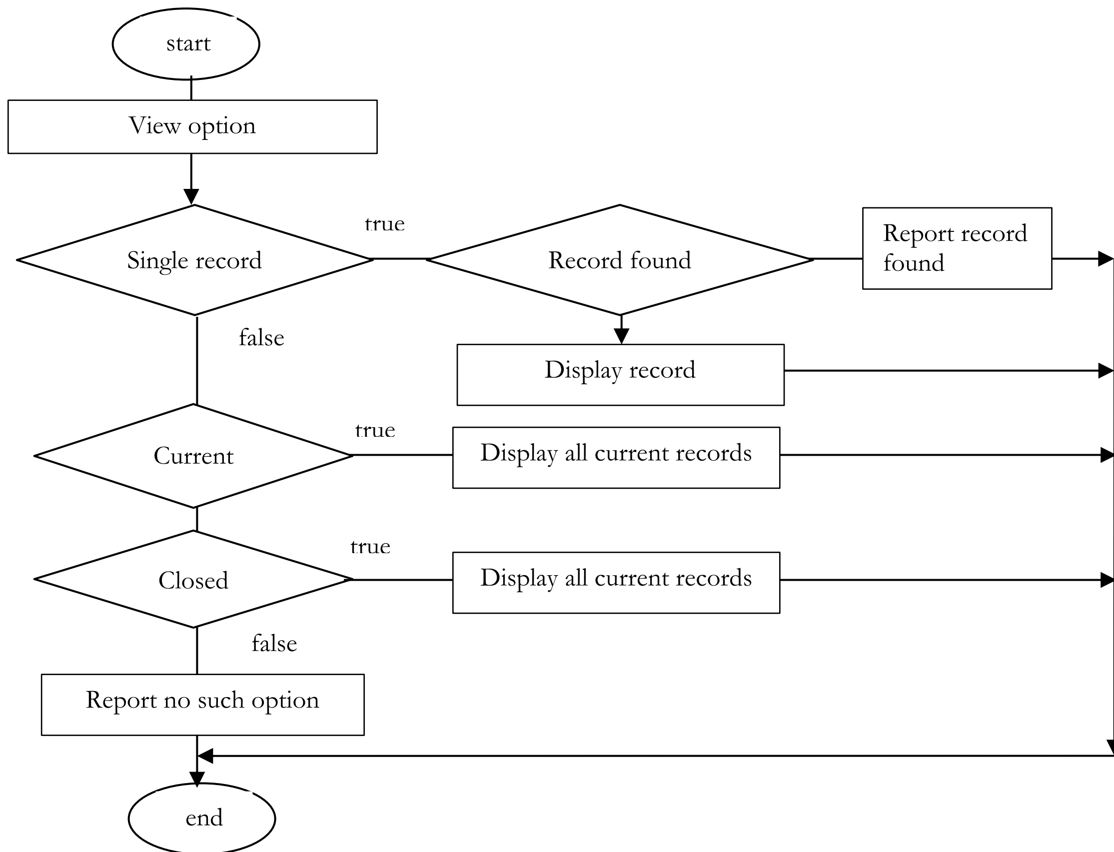
The third option, the updating of an account, runs parallel to removing an account, in that the account in question must first be located. The difference however is that if the account is located it is not removed, instead we must ascertain from the user whether the update is a deposit, or a withdrawal. Deposit is quite simple; this is just a matter of getting the amount representing the deposit and adding it to the existing balance, to form a new balance. Withdrawal other the other hand requires determining if there is sufficient funds in the account to effect the withdrawal. Finally, if the account does not exist in the database, the user should be alerted. See **Figure 1.23** for an illustration.

Figure 1.23 Updating an account



The displaying of information is summarized in **Figure 1.24**, where the information to be displayed is either about a single customer, a list of the current customers, or a list of the closed accounts.

Figure 1.24 Displaying account information



Listing 1.6 shows the class GetData that will be used to input the data.

Listing 1.6

```
1.  import javax.swing.JOptionPane;
2.  class GetData
3.  {
4.          public static double getDouble(String s)
5.          {
6.              return Double.parseDouble(getWord(s));
7.          }
8.          public static int getInt(String s)
9.          {
10.             return Integer.parseInt(getWord(s));
11.         }
12.         public static String getWord(String s)
13.         {
14.             return JOptionPane.showInputDialog(s);
15.         }
16. }
```

Listing 1.7 shows the test class called TestBankAccount which creates the database objects for storing the current customers, and the closed accounts. See **Lines 15** and **16** respectively. The outer while loop which spans **Lines 22** thru **159** keeps the application running until the boolean variable established on **Line 20** is set to **true**. The menu, defined on **Line 24**, allows the user to select the option of creating a new account, updating an existing account, closing an account, viewing account information, or terminating the application. The outer switch statement beginning on Line 26 allows the user to make the selection. The first case **Lines 28** thru **51** creates a new account and adds it to the database of active accounts. The second option case 2, **Lines 52** thru **80** allows the user to update an account, if possible. Option 3 which spans **Lines 81** thru **93** shows how an account is removed from the active list and get stored into the list of closed accounts, where possible. Option 4 which spans **Lines 94** thru **151** determines which account or set of accounts is displayed. Finally, option 5 which spans **Lines 153** thru **156** terminates the application.

Listing 1.7 The test class TestBank.java

```
1.	package BankAccount;
2.
3.	import java.text.DateFormat;
4.	import java.util.Date;
5.	import java.text.NumberFormat;
6.	import javax.swing.JOptionPane;
7.	import javax.swing.JTextArea;
8.	import javax.swing.JScrollPane;
9.	import java.util.ArrayList;
10.
11.	public class TestBankAccount
12.	{
13.		public static void main(String args[])
14.		{
15.			Database db = new Database();  // Creating database for active accounts
16.			Database close = new Database();   // Creating database for inactive accounts
17.			DateFormat df = DateFormat.getDateInstance(DateFormat.LONG);
18.			Date now = new Date();
19.			NumberFormat nf = NumberFormat.getCurrencyInstance();
20.			boolean done = false;
21.
22.			while (!done)
23.			{
24.				int menu = GetData.getInt("\tUnited Bank of Java\n" + "\t" + df.format(now) + "\n"
25.					+ "\nPlease Choose From the Following:" + "\n1. Create New Account\n2. Update
26.					Existing Account Account "+ "\n3. Close an Account\n4. View Account Information\n5.
27.					Exit");
28.				switch(menu)
29.				{
30.					case 1: //Creating a BankAccount object and storing it in the database
31.					// Creating Name object
32.					String f = GetData.getString("Enter First Name") ;
```

```
33.                 String l = GetData.getString("Enter Last Name") ;
34.                 Name n = new Name(f,l);
35.
36.                 // Creating Address object
37.                 String str = GetData.getString("Enter Street Name") ;
38.                 String city = GetData.getString("Enter City") ;
39.                 String st = GetData.getString("Enter State") ;
40.                 String zip = GetData.getString("Enter Zip") ;
41.
42.                 // Creating Customer object
43.                 Address addr = new Address(str,city,st,zip);
44.                 String accNo = GetData.getString("Enter Account Number") ;
45.                 Customer c = new Customer(n,accNo,addr);
46.
47.             // Creating BankAccount object
48.             double amount = GetData.getDouble("Enter First Deposit") ;
49.             BankAccount ba = new BankAccount(c, amount);
50.
51.             // Add BankAccount object to the database
52.             db.add(ba);
53.         break;
54.         case 2: //Update Account
55.             accNo = GetData.getString("Enter Account Number of Account you'd like to update")  ;
56.             db.search(accNo);
57.             if (!db.inList())
58.                 JOptionPane.showMessageDialog(null, "Account not found.");
59.             else
60.             {
61.                 int option = GetData.getInt("Would you like to (1) Deposit, (2) Withdraw");
62.                 switch(option)
63.                 {
64.                     case 1:
65.                         double amt = GetData.getDouble("Enter amount you'd like to deposit");
66.                         BankAccount b = db.getAccount();
67.                         b.deposit(amt);
68.                     break;
69.                     case 2:
70.                         double amnt = GetData.getDouble("Enter amount you'd like to withdraw") ;
71.                         BankAccount bnk = db.getAccount();
72.                         if (!bnk.isSufficient(amnt))
73.                             JOptionPane.showMessageDialog(null, "Insufficient funds, withdrawal cannot
                            be done.");
74.                         else
75.                             bnk.withdraw(amnt);
76.                     break;
77.                     default:
78.                         JOptionPane.showMessageDialog(null, "Invalid selection. To return to main menu,
                        please deposit or withdraw $0");
79.                     break;
80.                 }
81.             }
82.         break;
```

```
83.         case 3: //Close Account
84.             accNo = GetData.getString("Cose account - Please enter Account No.)");
85.             db.search(accNo);
86.             if (!db.inList())
87.                 JOptionPane.showMessageDialog(null, "Account not found.");
88.             else
89.             {
90.                 BankAccount b = db.getAccount();
91.                 int index = db.getIndex();
92.                 db.add( db.delete(index) );
93.                 JOptionPane.showMessageDialog(null, "The Account " + accNo + " has been closed.");
94.             }
95.         break;
96.         case 4: //View Account
97.             int view = GetData.getInt("What information would you like to view?\n1. Single account\n2. All
            active accounts\n3. All inactive accounts\n");
98.
99.             switch(view)
100.            {
101.                case 1:  // View a single account
102.                    accNo = GetData.getString("View – account. Please enter Account No.");
103.                    db.search(accNo);
104.                    if(!db.inList())
105.                        JOptionPane.showMessageDialog(null, "Account not found.");
106.                    else
107.                    {
108.                       BankAccount bb = db.getAccount();
109.                       String s = "Customer\t" + bb.getCustomer().getName().getFirst() + "\t" +
                       bb.getAmount() ;
110.                       JOptionPane.showMessageDialog(null, s, "Bank Account " +
                       bb.getCustomer().getAccountNumber(),
                       JOptionPane.INFORMATION_MESSAGE);
111.                    }
112.                break;
113.                case 2: // View all active accounts
114.                    ArrayList list = db.getList();
115.                    if(list.isEmpty())
116.                        JOptionPane.showMessageDialog(null, "List is empty");
117.                    else
118.                    {
119.                        int i = 0, length = db.size();
120.                        String s = "";
121.                        while(i < length)
122.                        {
123.                            BankAccount b = (BankAccount)list.get(i);
124.                            s = s + "Customer Name: " + b.getCustomer().getName().getFirst() + "  " +
                            b.getCustomer().getName().getLast() + "\nAccount number: " +
                            b.getCustomer().getAccountNumber() + "\n"
125.                            + b.getCustomer().getAddress().getStreet() + ", " +
                            b.getCustomer().getAddress().getCity() + ", " +
                            b.getCustomer().getAddress().getState() + ", "
126.                            + b.getCustomer().getAddress().getZip() + "\n" + nf.format(b.getAmount()) +
```

```
                                        "\n";
127.                                    i++;
128.                               }
129.                               display(s, "Active Accounts", JOptionPane.INFORMATION_MESSAGE);
130.                          }
131.                          break;
132.                               case 3: // View all closed accounts
133.                                   ArrayList closed = db.getList();
134.
135.                                   if(closed.isEmpty())
136.                                       JOptionPane.showMessageDialog(null, "List is empty");
137.                                   else
138.                                   {
139.                                       int i = 0, length = db.getSize();
140.                                       String s = "";
141.                                       while(i < length)
142.                                       {
143.                                           BankAccount b = (BankAccount)closed.get(i);
144.                                           s = s + "Name " + b.getCustomer().getName().getFirst() + "  " +
                                               b.getCustomer().getName().getLast() + "\tAccount number: " +
                                               b.getCustomer().getAccountNumber() + "\n";
145.                                           i++;
146.                                       }
147.                                       display(s, "Closed Accounts", JOptionPane.INFORMATION_MESSAGE);
148.                                   }
149.                               break;
150.                                   default:
151.                                  JOptionPane.showMessageDialog(null, "Invalid option.");
152.                                   break;
153.                        }// End view
154.                 break;
155.                 case 5: //Exit
156.                             done = true;
157.                 break;
158.                 default:
159.                             JOptionPane.showMessageDialog(null, "Account not found.");
160.                 break;
161.               }
162.             }
163.         }
164.         static void display(String s, String heading, int MESSAGE_TYPE)
165.         {
166.             JTextArea text = new JTextArea(s, 20, 30);
167.             JScrollPane pane = new  JScrollPane(text);
168.             JOptionPane.showMessageDialog(null, pane, heading, MESSAGE_TYPE);
169.         }
170.  }
```

Self-Check

1. What is meant by the term coupling? How is it difference from cohesion?

2. Write a class called Dealer that has fields of type Vehicle and Customer. The field for Vehicle is a class that has fields make, model, and VIN. The field Customer has fields Name and account number. The field called Name is also an existing class with fields - last name and first name.

   (a) Define the class Dealer so that a Dealer object can be created either by a customer object alone, or by a customer object and a vehicle object.
   (b) Provide accessor methods for each type of fields in the class Dealer.
   (c) Provide a mutator method that changes the vehicle object. (A test class is not necessary).
   (d) Suppose the class Name has a method getLastName(), that returns the last name of a customer. If the reference variable **deal** is an instance of Dealer, write appropriate Java code that will extract and return the last name of a customer.

Chapter Summary

- Object-oriented design (OOD) is the philosophy of developing an object-oriented model of a software system, by defining the classes and their interactions with one another.

- The major benefits of OOD are easy maintainable systems, easy understandable systems, easy expandable systems, and reusable components of a system.

- The Unified Modeling Language (**UML**) uses a set of diagrams to model objects. This language features several types of diagrams, among which is the class diagram.

- Class diagrams are of four types, two of which are dependency diagram and association diagram.

Programming Exercises

1. The establishment called ABC Enterprise requires a Java program to keep a database of the inventory of the products that it sells. Each product is identified by its manufacturer, its name, the quantity, and unit price. Note: a manufacturer is characterized by its company's name and address
   In addition to storing the information, the program should be able to make updates to the quantity and/or the price as time goes on. That is, when a sale is made, the quantity of that product must be reduced; similarly, when a product is re-ordered, the quantity of that product must be increased. Also, when there is a change in the price of a product, the price must be changed. The change must be interpreted as a replacement of the value. New products may be added to the inventory at any time; also, a product may be removed from the inventory at any time. Maintain a separate list the products that have been deleted from the database of active products.

Your program must be able to produce three kinds of reports, namely:
Locate a single product and display its name, price and quantity alone.
The inventory report should be structured as follows:

Product	Purchase Date	Quantity	Price	Manufacturer	State
Telephone	01/20/2013	10	254.99	Motorola	FL
Computer	01/06/2013	15	756.99	CBS	NY
:	:	:	:	:	:
:	:	:	:	:	:

The list of deleted products should be structured as follows:

Product	Date	Manufacturer
Paper reams	01/20/2013	Morgan Jewelry
:	:	:

In your design, convince yourself that you need a minimum of four classes, not including the test class – Product, Manufacturer, Address, and Database. You may use the class called GetData.java, **Listing 1.6**, for inputting the data. Use a scrollable pane to display your output.

2. Imagine that you were required to write a Java program which will store, manipulate, and print student registration information.

As part of the solution, identify the following classes:

(a) Student
(b) Admissions.

The class Student has the following fields – Name, Address, Id number, and Date, where:

(a) Name is a user defined class comprising of at minimum first name and last name.
(b) Address is a user defined class comprising of fields - street, city, state, and zip code.
(c) Date is a predefined class in the java.util package
(d) Id number a string variable that uniquely identifies a student.

The class Admissions stores and manipulates the student information (student record). Because the list of students grows dynamically, it is best to use a dynamic data structure such as the ArrayList to store the information. This class should do the following, among other possible activities:

(a) Add student to the list
(b) Remove student from the list. This would first involve locating the record in order to remove it. In order to determine which record to remove you must supply the Id number as the search argument.

You are to provide a test class that coordinates the activities of the classes outlined above, by:

- Creating student objects and adding them to the database of the Admissions object
- Removing a student from the database
- Change a student's last name
- Displaying list of currently registered students
- Displaying list of all students that were dropped from the course

The output must be formatted as follows, and must be placed in a scrollable pane.

CURRENTLY ENROLLED
Id number: 123456
Name: Williams, John
Address: 2525 Hartsfield Road
Tallahassee, FL 33319
Date: September 5, 2010
:
:
STUDENT WHO WERE DROPPED
Id number: 56789-0
Name: Roberts, Kay-Anne
Date: September 5, 2010
:
:

3. Write a Java program which will store, manipulate, and print student registration information.

As part of the solution, identify the following classes:
(c) Student
(d) Admissions.

The class Student must have the following fields – Name, Address, Id number, Courses, and Date, where:
(e) Name is a user defined class comprising of at minimum first name and last name.
(f) Address is a user defined class comprising of fields - street, city, state, and zip code.
(g) Date is a predefined class in the java.util package
(h) The field Courses is a set of no more than five (5) string values representing the courses being registered for. Course names supplied are assumed to be valid and contains no blank space, for instance COP3804 is valid but not COP 3804.
(i) Id number a string variable that uniquely identifies a student.
The class Student must be capable of adding courses and dropping courses

The class Admissions stores and manipulates the student information (student record). Because the list of students grows dynamically, it is best to use a dynamic data structure such as the ArrayList to store the information. This class should do the following, among other possible activities:
(c) Add student to the list
(d) Remove student from the list, which would first involve locating the record in order to remove it. In order to determine which record to remove you must supply the Id number as the search argument.

You are to provide a test class that coordinates the activities of the classes outlined above, by:
- Creating student objects and adding them to the database of the Admissions object

- Manipulating student record by:
    - Adding a course(s)
    - Dropping a course(s)
- Removing a student from the database
- Displaying list of currently registered students
- Displaying list of all students that were dropped from the course

The output must be formatted as follows:

CURRENTLY ENROLLED

Id number:	123456
Name:	Williams, John
Address:	2525 Hartsfield Road Tallahassee, FL 33319
Date:	September 5, 2009
Courses:	COP3804, MATH2050, ENG3300

:
:

STUDENT WHO WERE DROPPED

Id number:	567890
Name:	Roberts, Kay-Anne
Date:	September 5, 2009

:
:

Note: Use the class GetData provided to enter the data from the keyboard.

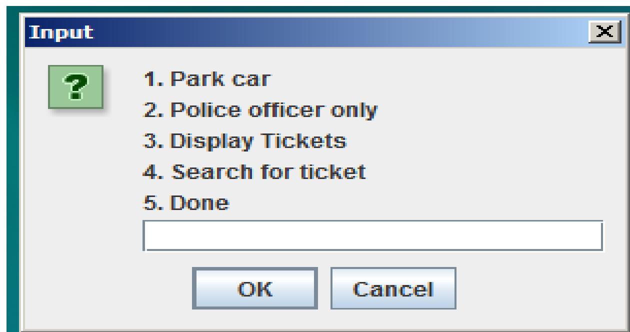
4. Parking Ticket Simulator

You are required to write a Java program to simulate police action re parked cars at parking meters. In modeling the solution, convince yourself that the program could have at least the following classes:

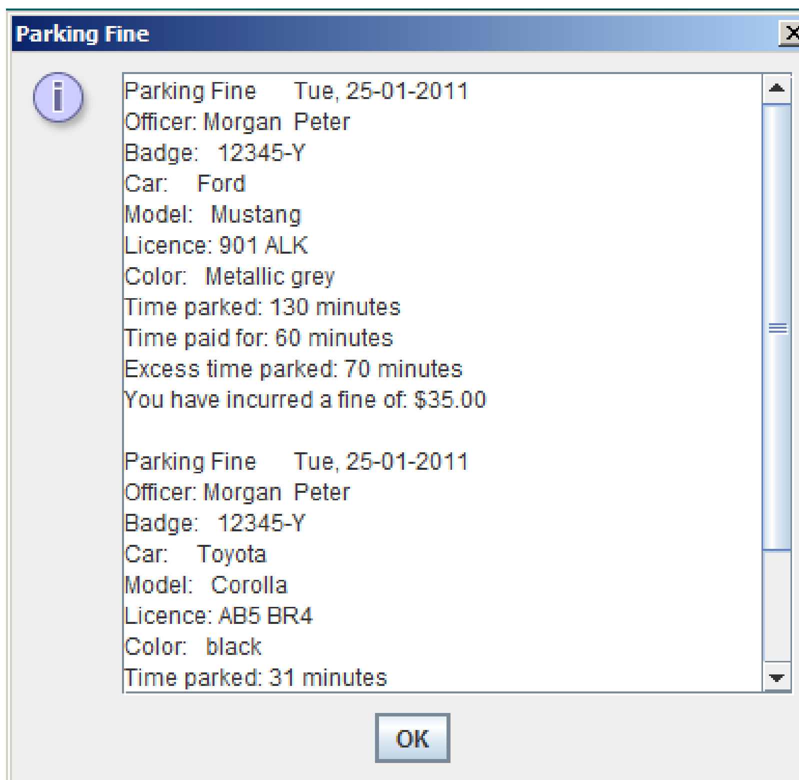
- A class that models a parked car. This class is characterized by a car object, and the number of minutes that the car has been parked. A car object is characterized by its make, model, color, and license number.
- A class that models a parking meter. This class is characterized by the number of minutes of parking time that has been purchased.
- A class that models a parking ticket. This class is characterized by, among other things, parked car objects and police officer objects. In addition it reports:
    - The make, model, color, license number of the illegally parked car.
    - The amount of fine. The amount of fine is determined as follows: $25.00 for the first hour, or part of an hour that the car is parked illegally, plus $10.00 for every additional hour or part of an hour that the car is parked illegally.
    - The name and badge of the officer issuing the ticket.
    - The date upon which the ticket was issued

- A class that models a police officer who is inspecting the parked car. This class is characterized by the following:
    - The officer's name and badge number. The name is an object characterized by at least first and last name.
    - The parked car object and the parking meter object to determine of the whether or not the car's time has expired
- A class that features storing, searching, and retrieving officer's copy of parking tickets.

Design a menu as shown below.



Your output must be displayed in a scrollable pane as shown below.



.