Chapter 11 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 11.1 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 11.1 UML Class diagram; class A depends on class B](image)

In Figure 11.1, 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 6.1 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.

```java
public class A {
    public void method1(B b) {
        //...
    }
    public void method2() {
        B b = new B();
        //...
    }
}
```

```java
public class B {
    public void method() {
        //...
    }
    public void method2() {
        //...
    }
}
```

**Listing 111.1 UML dependency relationship between class A and class B**

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.

<table>
<thead>
<tr>
<th>Frequency</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0..1</td>
<td>Zero or one time</td>
</tr>
<tr>
<td>1</td>
<td>Only once</td>
</tr>
<tr>
<td>0..*</td>
<td>Zero or more times</td>
</tr>
<tr>
<td>1..*</td>
<td>1 or more times</td>
</tr>
<tr>
<td>n</td>
<td>Only n times, n &gt; 1</td>
</tr>
<tr>
<td>0..n</td>
<td>Zero or more times, where n &gt; 1</td>
</tr>
<tr>
<td>1..n</td>
<td>1 or more times, where n &gt; 1</td>
</tr>
</tbody>
</table>
Figure 11.2 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.

![UML Class diagram; class A may depend on class B multiple times](image)

**Association Diagram**

A class association diagram defines a relationship that is much stronger than dependency relationship. Figure 11.3 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.

![A UML Association relationship diagram](image)

**Example 11.1** Let us consider the classes Person and Name as shown in Figure 11.4. 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.

![Association diagram of class Person with respect to class Name](image)
If we were to remove the field \textit{name} from the class Person, then the class Person, having just the field, \textit{phone}, would not establish that we are talking about person. Hence by including the field, \textit{name} to Person, at least intuitively adds the meaning person. Figure 11.5 shows the class Person without the field \textit{name}.

\begin{table}[h]
\centering
\begin{tabular}{|l|}
\hline
\textbf{Person} \\
\hline
\>phone: String \\
\hline
\>+Person(phone: String) \\
\>+getPhone(): String \\
\hline
\end{tabular}
\caption{class Person without the field name}
\end{table}

\section*{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.

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.

Figure 11.6 A software system of six components
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 11.6 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. 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](image)

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 11.8.

![Figure 11.8 Association relationship between BankAccount and Customer](image)

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.
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 11.6, 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 11.10. 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.

<table>
<thead>
<tr>
<th>Customer</th>
</tr>
</thead>
<tbody>
<tr>
<td>-name : Name</td>
</tr>
<tr>
<td>-addr : Address</td>
</tr>
<tr>
<td>-acc_no : String</td>
</tr>
</tbody>
</table>

+Customer(n: Name, a: Address, acc: String)
+getName(): Name
+getAddress(): Address
+getAccountNumber(): String
+changeAccountNumber(acc: String): void

Figure 11.10 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 11.10, 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.

```
public class Customer {
    private Name name;
    private String acc_no;
    private Address address;

    public Customer (Name n, String ac, Address addr) {
        name = n;
        acc_no = ac;
        this.address = addr;
    }

    public Name getName() {
        return name;
    }

    public Address getAddress() {
        return address;
    }

    public String getAccountNumber() {
        return acc_no;
    }

    void changeAccountNumber(String acc) {
        acc_no = acc;
    }
}
```

Listing 1.2 Class Customer

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.
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 1.11 The entity BankAccount

<table>
<thead>
<tr>
<th>BankAccount</th>
</tr>
</thead>
<tbody>
<tr>
<td>-cust : Customer</td>
</tr>
<tr>
<td>-balance : double</td>
</tr>
<tr>
<td>+BankAccount(Customer cmr: Name, ibal: double)</td>
</tr>
<tr>
<td>+getCustomer(): Customer</td>
</tr>
<tr>
<td>+getBalance(): double</td>
</tr>
<tr>
<td>+deposit (amt: double): void</td>
</tr>
<tr>
<td>+withdraw(amt: double): void</td>
</tr>
</tbody>
</table>

Figure 1.12 The entity BankAccount

<table>
<thead>
<tr>
<th>BankAccount</th>
</tr>
</thead>
<tbody>
<tr>
<td>-cust : Customer</td>
</tr>
<tr>
<td>-balance : double</td>
</tr>
<tr>
<td>+BankAccount(Customer cmr: Name, ibal: double)</td>
</tr>
<tr>
<td>+getCustomer(): Customer</td>
</tr>
<tr>
<td>+getBalance(): double</td>
</tr>
<tr>
<td>+deposit (amt: double): void</td>
</tr>
<tr>
<td>+withdraw(amt: double): void</td>
</tr>
<tr>
<td>+isSufficient(amt: double): boolean</td>
</tr>
</tbody>
</table>
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 there is one). 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 11.13.

**Figure 11.13** 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.

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 11.13.
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.

<table>
<thead>
<tr>
<th>Database</th>
</tr>
</thead>
<tbody>
<tr>
<td>-list : ArrayList</td>
</tr>
<tr>
<td>-account : BankAccount</td>
</tr>
<tr>
<td>+ Database ()</td>
</tr>
<tr>
<td>+add(account: BankAccount): void</td>
</tr>
<tr>
<td>+delete(accountNumber: String): BankAccount</td>
</tr>
<tr>
<td>+search (accountNumber: String): int</td>
</tr>
<tr>
<td>+getAccount(accountNumber: String): BankAccount</td>
</tr>
</tbody>
</table>

**Figure 11.15 The entity Database**

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.

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

\(^1\) For an expandable list, the class ArrayList is more appropriate than an array which is non-expandable.
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 11.16 shows a more cohesive class than the one shown in Figure 11.15.

**Figure 11.16 The entity Database**

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 11.17. A highly cohesive system generally gives rise to a lowly coupled system.
The graph of a cohesive system generally forms a perfect tree.

Figure 11.17