# **Software Engineering - I**

An Introduction to Software Construction Techniques for Industrial Strength Software

Chapter 6 – Software Design

#### Introduction

Recalling our discussion of software construction process, once the requirements of a software system have been established, we proceed to design that system. During the design phase, the focus shifts from what to how. That is, at this stage we try to answer the question of how to build the system. The objective of the design process is to analyze and understand the system in detail so that features and constituent components of at least one feasible solution are identified and documented. The design activity provides a roadmap to progressively transform the requirements through a number on stages into the final product by describing the structure of the system to be implemented.

It includes modeling of the data structures and entities, the physical and logical partitioning of the system into components, and the interfaces between different components of the system as well as interfaces to the outside world. Sometimes design of algorithms is also included in this activity.

#### **Managing Complexity of a Software System**

A complex system that works is invariably found to have evolved from a simple system that worked. The structure of a system also plays a very important role. It is likely that we understand only those systems that have hierarchical structure and where intracomponent linkages are generally stronger than inter component linkages. To manage the complexity of the system we need to apply the principles of separation of concern, modularity, and abstraction. This leads to designs that are easy to understand and hence easy to maintain.

Separation of concern, modularity, and abstraction are different but related principles.

Separation of concern allows us to deal with different individual aspects of a problem by considering these aspects in isolation and independent of each other.

A complex system may be divided into smaller pieces of lesser complexity called modules. This is the classic divide-and-conquer philosophy – if you cannot solve a complex problem, try to break it into smaller problems that you can solve separately and then integrate them together in a systematic fashion to solve the original problem. One major advantage of modularity is that it allows the designer to apply the principle of separation of concern on individual modules.

#### **Software Design Process**

Software design is not a sequential process. Design of a software system evolves through a number of iterations. The design process usually involves developing a number of different models, looking at the system from different angles and describing the system at various levels of abstraction. Like the various different models used during requirement engineering domain models, these models complement each other. As stated earlier,

software design provides a road map for implementation by clearly describing how the software system is to be realized.

A activities performed at this stage include design of the software architecture by showing the division of system into sub-systems or modules, the specification of the services provided by these sub-systems and their interfaces with each other, division of each sub-system into smaller components and services and interfaces provided by each one of these components. Data modeling is also an essential activity performed during the design phase. This includes the identification of data entities and their attributes, relationships among these entities, and the appropriate data structures for managing this data.

#### **Software Design Strategies**

Software design process revolves around decomposing the system into smaller and simpler units and then systematically integrate these units to achieve the desired results. Two fundamental strategies have been used to that end. These are functional or structured design and object oriented design.

In the functional design, the structure of the system revolves around functions. The entire system is abstracted as a function that provides the desired functionality (for example, the main function of a C program). This main function is decomposed into smaller functions and it delegates its responsibilities to these smaller functions and makes calls to these functions to attain the desired goal. Each of these smaller functions is decomposed into even smaller functions if needed. The process continues till the functions are defined at a level of granularity where these functions can be implemented easily. In this design approach, the system state, that is the data maintained by the system, is centralized and is shared by these functions.

The object-oriented design takes a different approach. In this case the system is decomposed into a set of objects that cooperate and coordinate with each other to implement the desired functionality. In this case the system state is decentralized and each object is held responsible for maintaining its own state. That is, the responsibility of marinating the system state is distributed and this responsibility is delegated to individual objects. The communication and coordination among objects is achieved through message passing where one object requests the other object if it needs any services from that object.

The object-oriented approach has gained popularity over the structured design approach during the last decade or so because, in general, it yields a design that is more maintainable than the design produced by the functional approach.

# **Software Design Qualities**

A software design can be looked at from different angles and different parameters can be used to measure and analyze its quality. These parameters include efficiency, compactness, reusability, and maintainability. A good design from one angle may not seem to be suitable when looked from a different perspective. For example, a design that yields efficient and compact code may not be very easy to maintain. In order to establish whether a particular design is good or not, we therefore have to look at the project and application requirements. For example, if we need to design an embedded system for the control of a nuclear reactor or a cruise missile, we would probably require a system that is very efficient and maintainability would be of secondary concern. On the other hand, in the case of an ordinary business system, we would have a reversal in priorities.

# **Maintainable Design**

Since, in general, maintenance contributes towards a major share of the overall software cost, the objective of the design activity, in most cases, is to produce a system that is easy to maintain. A maintainable design is the one in which cost of system change is minimal and is flexible enough so that it can be easily adapted to modify exiting functionality and add new functionality.

In order to make a design that is maintainable, it should be understandable and the changes should be local in effect. That is, it should be such that a change in some part of the system should not affect other parts of the system. This is achieved by applying the principles of modularity, abstraction, and separation of concern. If applied properly, these principles yield a design that is said to be more cohesive and loosely coupled and thus is easy to maintain.

# **Coupling and Cohesion**

Coupling is a measure of independence of a module or component. Loose coupling means that different system components have loose or less reliance upon each other. Hence, changes in one component would have a limited affect on other components.

Strong cohesion implies that all parts of a component should have a close logical relationship with each other. That means, in the case some kind of change is required in the software, all the related pieces are found at one place. Hence, once again, the scope is limited to that component itself.

A component should implement a single concept or a single logical entity. All the parts of a component should be related to each other and should be necessary for implementing that component. If a component includes parts that are not related to its functionality, then the component is said to have low cohesion.

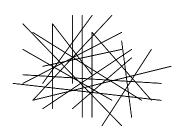
Coupling and cohesion are contrasting concepts but are indirectly related to each other. Cohesion is an internal property of a module whereas coupling is its relationship with other modules. Cohesion describes the intra-component linkages while couple shows the

inter-component linkages. Coupling measures the interdependence of two modules while cohesion measures the independence of a module. If modules are more independent, they will be less dependent upon others. Therefore, a highly cohesive system also implies less coupling.

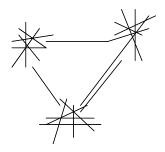
A good example of a system with a very high cohesion and very less (almost nil) coupling is the electric subsystem of a house that is made up of electrical appliances and wires. Since each one of the appliances has a clearly definable function that is completely encapsulated within the appliance. That means that an appliance does not depend upon any other appliance for its function. Therefore, each appliance is a highly cohesive unit. Since there are no linkages between different appliances, they are not coupled. Let us now assume that we have added a new centralized control unit in the system to control different appliances such as lights, air conditioning, and heating, according to certain settings. Since this control unit is dependent upon the appliances, the overall system has more coupling than the first one.

Modules with high cohesion and low coupling can be treated and analyzed as black boxes. This approach therefore allows us to analyze these boxes independent of other modules by applying the principle of separation of concern.

Coupling and cohesion can be represented graphically as follows.



**High Coupling** 

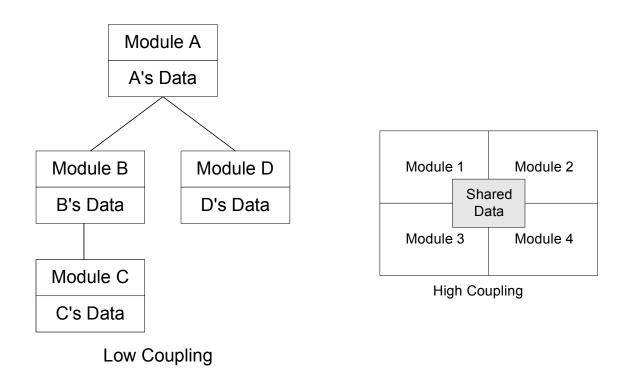


Low Coupling

This diagram depicts two systems, one with high coupling and the other one with low coupling. The lines depict linkages between different components. In the case of highly coupled system, module boundaries are not well defined, as everything seems to be connected with everything else. On the other hand, in the system with low coupling modules can be identified easily. In this case intra component linkages are stronger while inter component linkages are weak.

#### **Example of Coupling**

The modules that interact with each other through message passing have low coupling while those who interact with each other through variables that maintain information about the state have high coupling. The following diagram shows examples of two such systems.



In order to understand this concept, let us consider the following example. In this example, we have a class vector in which the data members have been put in the public part.

```
class vector {
   public:
     float x;
     float y;
     vector (float x, float y);
     float getX();
     float getY();
     float getMagnitude();
     float getAngle();
};
```

Now let us assume that we want to write a function to calculate dot product of two vectors. We write the following function.

```
float myDotProduct1(vector a, vector b)
{
    float temp1 = a.getX() * b.getX();
    float temp2 = a.getY() * b.getY();
    return temp1 + temp2;
}
```

Since the data members are public, one could be enticed to use these members directly (presumably saving some function calls overhead) and rewrite the same function as follows:

```
float myDotProduct2(vector a, vector b)
{
    float temp1 = a.x * b.x;
    float temp2 = a.y * b.y;
    return temp1 + temp2;
}
```

So far, there does not seem to be any issue. But the scenario changes as soon as there are changes in the class implementation. Now let us assume that for some reason the class designer changes the implementation and data structure and decides to stores the angle and magnitude instead of the x and y components of the vector. The new class looks like as follows:

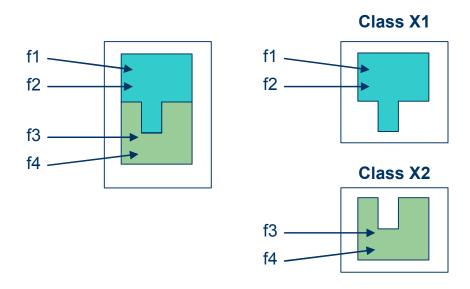
```
class vector {
   public:
     float magnitude;
     float angle;
     vector (float x, float y);
     vector (float magnitude, float angle);
     float getX();
     float getY();
     float getAngle();
};
```

Now we see the difference in the two implementations of the dot product function written by the user of this class. In the first case, as the dot product function is dependent upon the public interface of the vector class, there will be no change while in the second case the function will have to be rewritten. This is because in the first case the system was loosely coupled while in the second case there was more dependency on the internal structure of the vector class and hence there was more coupling.

# **Example of Cohesion**

As mentioned earlier, strong cohesion implies that all parts of a component should have a close logical relationship with each other. That means, in case some kind of change is required in the software, all the related pieces are found at one place.

A class will be cohesive if most of the methods defined in a class use most of the data members most of the time. If we find different subsets of data within the same class being manipulated by separate groups of functions then the class is not cohesive and should be broken down as shown below.



As an example, consider the following order class:

```
class order {
   public:
     int getOrderID();
     date getOrderDate();
     float getTotalPrice();
     int getCustometId();
     string getCustomerName();
     string getCustometAddress();
     int getCustometPhone();
     void setOrderID(int oId);
     void setOrderDate(date oDate);
     void setTotalPrice(float tPrice);
     void setCustometId(int cId);
     void setCustomerName(string cName);
     void setCustometAddress(string cAddress);
     void setCustometPhone(int cPhone);
     void setCustomerFax(int cFax)
   private:
     int oredrId;
     date orderDate;
     float totalPrice;
     item lineItems[20];
     int customerId;
     string customerName;
     int customerPhone;
     int customerFax;
};
```

The Order class shown above represents an Order entity that contains the attributes and behavior of a specific order. It is easy to see that this contains information about the order as well as the customer which is a distinct entity. Hence it is not a cohesive class and must be broken down into two separate classes as shown. In this case each on of these is a more cohesive class.

```
class order {
    public:
     int getOrderID();
     date getOrderDate();
     float getTotalPrice();
     int getCustometId();
     void setOrderID(int oId);
     void setOrderDate(date oDate);
     void setTotalPrice(float tPrice);
     void setCustometId(int cId);
     void addLineItem(item anItem);
    private:
     int oredrId;
     date orderDate;
     float totalPrice;
     item lineItems[20];
     int customerId;
};
class customer {
    public:
     int getCustometId();
     string getCustomerName();
     string getCustometAddress();
     int getCustometPhone();
     int getCustomerFax();
     void setCustometId(int cId);
     void setCustomerName(string cName);
     svoid setCustometAddress(string cAddress);
     void setCustometPhone(int cPhone);
     void setCustomerFax(int cFax)
    private:
     int customerId;
     string customerName;
     int customerPhone;
     int customerFax;
};
```

# **Abstraction and Encapsulation**

Abstractions is a technique in which we construct a model of an entity based upon its essential characteristics and ignore the inessential details. The principle of abstraction also helps us in handling the inherent complexity of a system by allowing us to look at its important external characteristic, at the same time, hiding its inner complexity. Hiding the internal details is called encapsulation. In fact, abstraction is a special case of separation of concern. In this case we separate the concern of users of the entity who only need to understand its external interface without bothering about its actual implementation.

Engineers of all fields, including computer science, have been practicing abstraction for mastering complexity. Consider the following example.

This function can be rewritten by abstracting out some of the logical steps into auxiliary functions. The new code is as follows.

```
viod swap(int &x, int &y)
{
    int temp;
    temp = x;
    x = y;
    y = temp;
}
```

```
int indexOfMinimumValue(int a[], int from, int to)
{
    int i, min;
    min = from;
    for (i = from+1; i < to; i++)
        if (a[i] < a[min]) min = i;
    return min;
}

void selectionSort(int a[], int size)
{
    int i, min;
    for (i = 0; i < size; i++)
    {
        min = indexOfMinimumValue(a, i, size);
        swap(a[i], a[min]);
    }
}</pre>
```

In this function we have abstracted out two logical steps performed in this functions. These functions are finding the index of the minimum value in the given range in an array and swapping the minimum value with the value at the ith index in the array. It is easy to see that the resultant new function is easier to understand than the previous version of the selection sort function. In the process, as a by-product, we have created two auxiliary function mentioned above, which are general in nature and hence can be used elsewhere as well. Principle of abstraction thus generates reusable self-contained components.

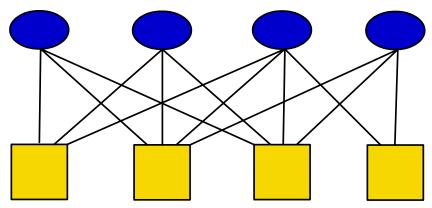
# **Function Oriented versus Object Oriented Design**

Let us now try to understand the difference between object-oriented and function oriented (or action oriented) approach.

In the case of action-oriented approach, data is decomposed according to functionality requirements. That is, decomposition revolves around function. In the OO approach, decomposition of a problem revolves around data. Action-oriented paradigm focuses only on the functionality of a system and typically ignores the data until it is required. Object-oriented paradigm focuses both on the functionality and the data at the same time. The basic difference between these two is decentralized control mechanism versus centralized control mechanism respectively. Decentralization gives OO the ability to handle essential complexity better than action-oriented approach.

This difference is elaborated with the help of the following diagram:

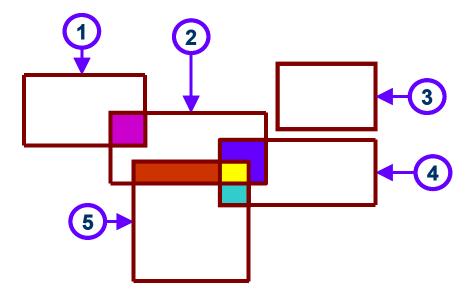
# **Functions**



#### **Data**

In this diagram, the ovals depict the function while rectangles/squares depict data. Since a function contains dynamic information while data contains only static information, if the function and data are managed separately, the required data components can be found by scanning a function but the functions that use a particular data cannot be found by just looking at the data. That is, the function knows about the data it needs to use but the data do not know about the functions using it. That means it is easy to make a change in a function since we would know which data components would be affected by this change. On the other hand, changing a data structure would be more difficult because it would not be easy to find all the functions that are using this data and hence also need to be modified.

Object oriented approach solves this problem by putting the relevant data and functionality together at one place. Hence, in case of a change, the effected components can be identified easily and the effect of change is localized. Therefore, maintenance becomes relatively easy as compared to function-oriented approach. This is made possible because the data is not shared in this case. Anyone needing any information contained in there would request the encapsulating object by sending it a message through the interface provided by the object. In this case we create highly cohesive objects by keeping the related data and function at one place and spinning-off non-related information into other classes. This can be elaborated with the help of the following diagram.



Let us assume that the circles represent sets of functions and rectangles represent data that these function use to carry out their operation. In the object-oriented design, the data areas that are common among different sets of functions would be spun-off into their own classes and the user function would use these data through their interfaces only. This is shown in the following diagram.

