**SOFTWARE DESIGN**

Design is a meaningful representation of something that is to be built. It can

be traced to a customer’s requirements and at the same time assessed for

quality against a set of predefined criteria for “good” design.

A set of design concepts has evolved over the years. According to M.A. Jackson,

“*The beginning of wisdom for a software engineer is to recognize the difference between*

*getting a program to work and getting it right.*” The various design concepts discussed

in this chapter provide the necessary framework for “getting it right.”

**5.1.1 Definition of Software Design**

The definitions of software design are as diverse as design methods. Some

important software design definitions are outlined below.

**According to Coad and Yourdon.** *Software Design is the practice of taking*

*a specification of externally observable behavior and adding details needed for actual*

*computer system implementation, including human interaction, task management, and*

*data management details*.

**117**

**118** SOFTWARE ENGINEERING AND TESTING

**According to Webster.** *In a sense, design is representation of an object being created.*

*A design information base that describes aspects of this object, and the design process can*

*be viewed as successive elaboration of representations, such as adding more information or*

*even backtracking and exploring alternatives.*

**According to Stevens.** *Software Design is the process of inventing and selecting*

*programs that meet the objectives for software systems.*

Input includes an understanding of the following:

\_ Requirements

\_ Environmental constraints

\_ Design criteria

The output of the design effort is composed of the following:

\_ Architecture design which shows how pieces are interrelated

\_ Specifications for any new pieces

\_ Definitions for any new data

**5.1.2 Design Objectives/Properties**

The various desirable properties or objectives of software design are:

1. ***Correctness.*** The design of a system is correct if the system built precisely

according to the design satisfies the requirements of that system. Clearly,

the goal during the design phase is to produce correct designs. However,

correctness is not the sole criterion during the design phase, as there can be

many correct designs. The goal of the design process is not simply to produce

a design for the system. Instead, the goal is to find the best possible design

within the limitations imposed by the requirements and the physical and

social environment in which the system will operate.

2. ***Verifiability.*** Design should be correct and it should be verified for correctness.

Verifiability is concerned with how easily the correctness of the design can be

checked. Various verification techniques should be easily applied to design.

3. ***Completeness.*** Completeness requires that all the different components of

the design should be verified, i.e., all the relevant data structures, modules,

external interfaces, and module interconnections are specified.

4. ***Traceability.*** Traceability is an important property that can get design

verification. It requires that the entire design element be traceable to the

requirements.

5. ***Efficiency.*** Efficiency of any system is concerned with the proper use of

scarce resources by the system. The need for efficiency arises due to cost

SYSTEM DESIGN **119**

considerations. If some resources are scarce and expensive, it is desirable that

those resources be used efficiently. In computer systems, the resources that

are most often considered for efficiency are processor time and memory. An

efficient system consumes less processor time and memory.

6. ***Simplicity.*** Simplicity is perhaps the most important quality criteria for

software systems. Maintenance of a software system is usually quite expensive.

The design of the system is one of the most important factors affecting the

maintainability of the system.

**Modularization**

Modularization is a technique to divide a software system into multiple discrete and independent modules, which are expected to be capable of carrying out task(s) independently. These modules may work as basic constructs for the entire software. Designers tend to design modules such that they can be executed and/or compiled separately and independently.

Modular design unintentionally follows the rules of ‘divide and conquer’ problem-solving strategy this is because there are many other benefits attached with the modular design of a software.

*DEPT OF CSE & IT VSSUT, Burla*

Advantage of modularization:

 Smaller components are easier to maintain

 Program can be divided based on functional aspects

 Desired level of abstraction ca n be brought in the program

 Components with high cohesion can be re-used again.

 Concurrent execution can be made possible

 Desired from security aspect

**Concurrency**

Back in time, all softwares were meant to be executed sequentially. By sequential execution we mean that the coded instruction will be executed one after another implying only one portion of program being activated at any given time. Say, a software has multiple modules, then only one of all the modules can be found active at any time of execution.

In software design, concurrency is implemented by splitting the software into multiple independent units of execution, like modules and executing them in parallel. In other words, concurrency provides capability to the software to execute more than one part of code in parallel to each other.

It is necessary for the programmers and designers to recognize those modules, which can be made parallel execution.

Example

The spell check feature in word processor is a module of software, which runs alongside the word processor itself.

**Coupling and Cohesion**

When a software program is modularized, its tasks are divided into several modules based on some characteristics. As we know, modules are set of instructions put together in order to achieve some tasks. They are though, considered as single entity but may refer to each other to work together. There are measures by which the quality of a design of modules and their interaction among them can be measured. These measures are called coupling and cohesion.

**Cohesion**

Cohesion is a measure that defines the degree of intra-dependability within elements of a module. The greater the cohesion, the better is the program design.

*DEPT OF CSE & IT VSSUT, Burla*

There are seven types of cohesion, namely –

 **Co-incidental cohesion -** It is unplanned and random cohesion, which might be the result of breaking the program into smaller modules for the sake of modularization. Because it is unplanned, it may serve confusion to the programmers and is generally not-accepted.

 **Logical cohesion -** When logically categorized elements are put together into a module, it is called logical cohesion.

 **Temporal Cohesion -** When elements of module are organized such that they are processed at a similar point in time, it is called temporal cohesion.

 **Procedural cohesion -** When elements of module are grouped together, which are executed sequentially in order to perform a task, it is called procedural cohesion.

 **Communicational cohesion -** When elements of module are grouped together, which are executed sequentially and work on same data (information), it is called communicational cohesion.

 **Sequential cohesion -** When elements of module are grouped because the output of one element serves as input to another and so on, it is called sequential cohesion.

 **Functional cohesion -** It is considered to be the highest degree of cohesion, and it is highly expected. Elements of module in functional cohesion are grouped because they all contribute to a single well-defined function. It can also be reused.

**Coupling**

Coupling is a measure that defines the level of inter-dependability among modules of a program. It tells at what level the modules interfere and interact with each other. The lower the coupling, the better the program.

There are five levels of coupling, namely -

 **Content coupling -** When a module can directly access or modify or refer to the content of another module, it is called content level coupling.

 **Common coupling-** When multiple modules have read and write access to some global data, it is called common or global coupling.

 **Control coupling-** Two modules are called control-coupled if one of them decides the function of the other module or changes its flow of execution.

 **Stamp coupling-** When multiple modules share common data structure and work on different part of it, it is called stamp coupling.

 **Data coupling-** Data coupling is when two modules interact with each other by means of passing data (as parameter). If a module passes data structure as parameter, then the receiving module should use all its components.

Ideally, no coupling is considered to be the best.

**COUPLING AND COHESION---B.B Agarwal Page no. 136**

e.

**Object Oriented Design**

Object oriented design works around the entities and their characteristics instead of functions involved in the software system. This design strategy focuses on entities and its characteristics. The whole concept of software solution revolves around the engaged entities.

Let us see the important concepts of Object Oriented Design:

 **Objects -** All entities involved in the solution design are known as objects. For example, person, banks, company and customers are treated as objects. Every entity has some attributes associated to it and has some methods to perform on the attributes.

 **Classes -** A class is a generalized description of an object. An object is an instance of a class. Class defines all the attributes, which an object can have and methods, which defines the functionality of the object.

*DEPT OF CSE & IT VSSUT, Burla*

In the solution design, attributes are stored as variables and functionalities are defined by means of methods or procedures.

 **Encapsulation -** In OOD, the attributes (data variables) and methods (operation on the data) are bundled together is called encapsulation. Encapsulation not only bundles important information of an object together, but also restricts access of the data and methods from the outside world. This is called information hiding.

 **Inheritance -** OOD allows similar classes to stack up in hierarchical manner where the lower or sub-classes can import, implement and re-use allowed variables and methods from their immediate super classes. This property of OOD is known as inheritance. This makes it easier to define specific class and to create generalized classes from specific ones.

 **Polymorphism -** OOD languages provide a mechanism where methods performing similar tasks but vary in arguments, can be assigned same name. This is called polymorphism, which allows a single interface performing tasks for different types. Depending upon how the function is invoked, respective portion of the code gets executed.

**Design Process**

Software design process can be perceived as series of well-defined steps. Though it varies according to design approach (function oriented or object oriented, yet It may have the following steps involved:

 A solution design is created from requirement or previous used system and/or system sequence diagram.

 Objects are identified and grouped into classes on behalf of similarity in attribute characteristics.

 Class hierarchy and relation among them are defined.

 Application framework is defined.

**Software Design Approaches**

There are two generic approaches for software designing:

**Top down Design**

We know that a system is composed of more than one sub-systems and it contains a number of components. Further, these sub-systems and components may have their one set of sub-system and components and creates hierarchical structure in the system.

Top-down design takes the whole software system as one entity and then decomposes it to achieve more than one sub-system or component based on some characteristics. Each sub-*DEPT OF CSE & IT VSSUT, Burla*

system or component is then treated as a system and decomposed further. This process keeps on running until the lowest level of system in the top-down hierarchy is achieved.

Top-down design starts with a generalized model of system and keeps on defining the more specific part of it. When all components are composed the whole system comes into existence.

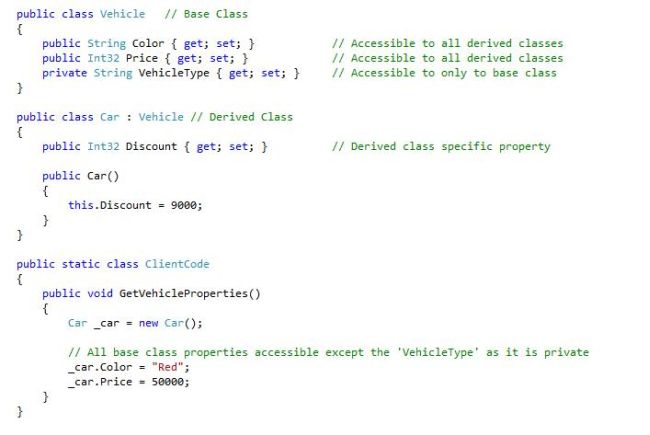
Top-down design is more suitable when the software solution needs to be designed from scratch and specific details are unknown.

**Bottom-up Design**

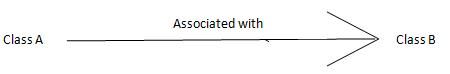
The bottom up design model starts with most specific and basic components. It proceeds with composing higher level of components by using basic or lower level components. It keeps creating higher level components until the desired system is not evolved as one single component. With each higher level, the amount of abstraction is increased.

Bottom-up strategy is more suitable when a system needs to be created from some existing system, where the basic primitives can be used in the newer system.

Both, top-down and bottom-up approaches are not practical individually. Instead, a good combination of both is used.

**Generalization**  
  
All of us know about the concept of inheritance. It is nothing but a kind of generalization. We define it simply with a base class having some properties, functions and so on. A new class will be derived from this base class and the child class will have access to all the functionality of the base or parent class (of course depending on the access modifiers defined on the base class members).  
  
It is also referred to as a "is-a-kind-of" or "is-a" relationship; that means that the child class is-a-kind-of a base class. This provides the ability to define common properties for derived classes in a common place and use of them depends on the derived class requirements.  
  
For example: we have a base class named Vehicle and a derived class named Car, from the base class. So Car will have access to all the functions, properties and so on of the base class Vehicle (depending on the access modifiers defined for base class members). See the code below:  
  
  
  
Here we have common properties like color and price in the base class that can be set depending on the derived class's requirements. Apart from these, it will add its specific property named Discount. But this class will not have access to the base class property VehicleType, since it is a private type. The same will be applicable for any other class that derives from the Vehicle class and any other member defined in the base class. Generalization is represented by the following symbol in UML:  
  
**Association**  
  
Association is defined as a structural relationship, that conceptually means that the two components are linked to each other. This kind of relation is also referred to as a using relationship, where one class instance uses the other class instance or vice-versa, or both may be using each other. But the main point is, the lifetime of the instances of the two classes are independent of each other and there is no ownership between two classes.  
  
For example, consider the same example of a Student-Teacher relationship. Conceptually speaking, each student can be associated with multiple teachers and each teacher can be associated with multiple students. Now to explain this relationship in terms of Object Oriented Programming, see the code below:  
  
  
  
Here, we have a single student instance "\_student". This student can have multiple teacher class instances associated with it, in other words "\_teacher1" and "\_teacher2".

1. First, both of these instances are being created outside the student class. So their lifetime is independent of the lifetime of the instance of the student. So even if "\_student" is disposed of explicitly, we can still have the teacher class instances as alive.
2. Secondly, any other student instance, say \_student2, can also have the same instances of teacher, in other words "\_teacher1" and "\_teacher2" associated with it. So we can also say that no instance of student is a parent of any instance of teacher. So there is no ownership of instances in the association.

Similarly, we can have the reverse case, where we have a teacher instance "\_teacher1" associated with "\_student1" and "\_student2", and so on. The same student instances can be associated with another teacher instance, say "\_teacher2".  
  
This association is represented by the following symbol in UML:  
  
  
  
So that described association. Now for the specialized cases of association, aggregation and composition.

* **Aggregation:**Aggregation is the same as association but with an additional point that there is an ownership of the instances, unlike association where there was no ownership of the instances. To understand it better, let's add another class named Department to our example explained above.

If we talk about the relation between Teacher and Department then conceptually, a Department can have multiple Teachers associated with it but each Teacher can belong to only one Department at a time. Now to explain this relationship in terms of Object Oriented Programming, see the code below:  
  
  
Here, we have a department instance "\_department1" and multiple instances of teacher, in other words "\_teacher1" and "\_teacher2".

1. First, the lifetime of "\_teacher1" and "\_teacher2" instances are independent of the lifetime of a "\_department1" instance, since they are instantiated outside the department class. So even if a "\_department1" instance is disposed if, "\_teacher1" and "\_teacher2" instances may continue to exist.
2. Secondly, "\_department1" can have multiple instances of teacher associated with it, but the reverse is not true, in other words "\_teacher1" and "\_teacher2" can belong to "\_department1" only. They cannot belong to any other instance like department2. So "\_department1" becomes the owner of the "\_teacher1" and "\_teacher2" instances. So here the ownership exists.

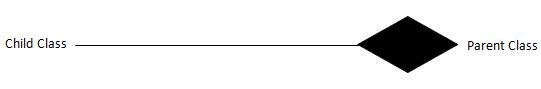
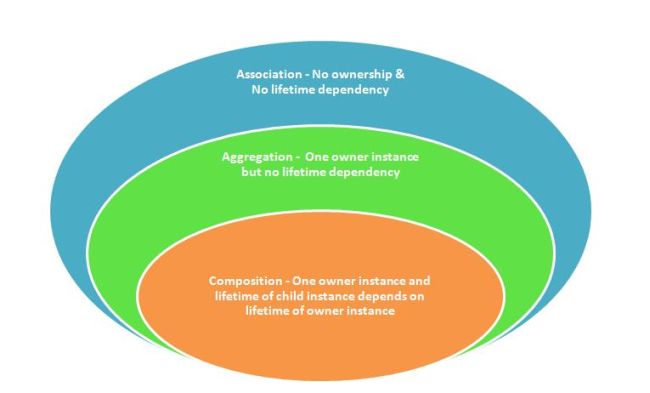
Aggregation is also referred to as a Weak Association and is represented by the following symbol in UML representation:



* **Composition:** This is the same as that of aggregation, but with the additional point that the lifetime of the child instance is dependent on the owner or the parent class instance. To the same code above, let's add another class named University.

So in this case also, conceptually, a university can have multiple departments in it. But each department can belong to a single university only. Now to explain this relationship in terms of Object Oriented Programming, see the code below:  


1. First, each department instance, in other words "\_department1" and "\_department2", can belong only to a single university instance at a time, \_university1. But a university instance can have multiple department instances attached to it. So this makes the "\_university1" instance the owner of the "\_department1" and "\_department2" instances. So this is the inherited feature of the aggregation in composition.
2. Secondly, here the lifetime of department instances are dependent on the instances of university, since they are being created inside the university class. So when the "\_university1" instance is disposed of the "\_department1" and "\_department2" instances are also killed. This is the composition concept.

Composition is also referred to as a Strong Association or Death relationship and is represented by the following symbol in UML representation:  
  
So if we closely observe the concepts of Association, Aggregation and Composition, we can say that composition is a subset of association and aggregation and aggregation is a subset of association. In other wors, Association is a super-set of aggregation and composition can be represented as:  
  
So this was all about these concepts and we can say that we were using these concepts in our programming but were not aware of their actual existence.