**SYSTEM DESIGN**

**INTRODUCTION**

Software design sits at the technical kernel of the software engineering process and is applied regardless of the development paradigm and area of application. Design is the first step in the development phase for any engineered product or system. The designer’s goal is to produce a model or representation of an entity that will later be built. Beginning, once system requirement have been specified and analyzed, system design is the first of the three technical activities -design, code and test that is required to build and verify software.

The importance can be stated with a single word “Quality”. Design is the place where quality is fostered in software development. Design provides us with representations of software that can assess for quality. Design is the only way that we can accurately translate a customer’s view into a finished software product or system. Software design serves as a foundation for all the software engineering steps that follow. Without a strong design we risk building an unstable system – one that will be difficult to test, one whose quality cannot be assessed until the last stage.

During design, progressive refinement of data structure, program structure, and procedural details are developed reviewed and documented. System design can be viewed from either technical or project management perspective. From the technical point of view, design is comprised of four activities – architectural design, data structure design, interface design and procedural design.

**NORMALIZATION**

It is a process of converting a relation to a standard form. The process is used to handle the problems that can arise due to data redundancy i.e. repetition of data in the database, maintain data integrity as well as handling problems that can arise due to insertion, updation, deletion anomalies.

Decomposing is the process of splitting relations into multiple relations to eliminate anomalies and maintain anomalies and maintain data integrity. To do this we use normal forms or rules for structuring relation.

**Insertion anomaly**: Inability to add data to the database due to absence of other data.

**Deletion anomaly**: Unintended loss of data due to deletion of other data.

**Update anomaly**: Data inconsistency resulting from data redundancy and partial update

**Normal Forms**: These are the rules for structuring relations that eliminate anomalies.

**FIRST NORMAL FORM**:

A relation is said to be in first normal form if the values in the relation are atomic for every attribute in the relation. By this we mean simply that no attribute value can be a set of values or, as it is sometimes expressed, a repeating group.

**SECOND NORMAL FORM**:

A relation is said to be in second Normal form is it is in first normal form and it should satisfy any one of the following rules.

1. Primary key is a not a composite primary key
2. No non key attributes are present
3. Every non key attribute is fully functionally dependent on full set of primary key.

**THIRD NORMAL FORM**:

A relation is said to be in third normal form if their exits no transitive dependencies.

**Transitive Dependency**: If two non key attributes depend on each other as well as on the primary key then they are said to be transitively dependent.

The above normalization principles were applied to decompose the data in multiple tables thereby making the data to be maintained in a consistent state.

**E – R DIAGRAMS**

* + The relation upon the system is structure through a conceptual ER-Diagram, which not only specifics the existential entities but also the standard relations through which the system exists and the cardinalities that are necessary for the system state to continue.
  + The Entity Relationship Diagram (ERD) depicts the relationship between the data objects. The ERD is the notation that is used to conduct the date modeling activity the attributes of each data object noted is the ERD can be described resign a data object descriptions.
  + The set of primary components that are identified by the ERD are
  + Data object
  + Relationships
  + Attributes
  + Various types of indicators.

The primary purpose of the ERD is to represent data objects and their relationships.



**DATA FLOW DIAGRAMS**

A data flow diagram is graphical tool used to describe and analyze movement of data through a system. These are the central tool and the basis from which the other components are developed. The transformation of data from input to output, through processed, may be described logically and independently of physical components associated with the system. These are known as the logical data flow diagrams. The physical data flow diagrams show the actual implements and movement of data between people, departments and workstations. A full description of a system actually consists of a set of data flow diagrams. Using two familiar notations Yourdon, Gane and Sarson notation develops the data flow diagrams. Each component in a DFD is labeled with a descriptive name. Process is further identified with a number that will be used for identification purpose. The development of DFD’S is done in several levels. Each process in lower level diagrams can be broken down into a more detailed DFD in the next level. The lop-level diagram is often called context diagram. It consist a single process bit, which plays vital role in studying the current system. The process in the context level diagram is exploded into other process at the first level DFD.

The idea behind the explosion of a process into more process is that understanding at one level of detail is exploded into greater detail at the next level. This is done until further explosion is necessary and an adequate amount of detail is described for analyst to understand the process.

Larry Constantine first developed the DFD as a way of expressing system requirements in a graphical from, this lead to the modular design.

A DFD is also known as a “bubble Chart” has the purpose of clarifying system requirements and identifying major transformations that will become programs in system design. So it is the starting point of the design to the lowest level of detail. A DFD consists of a series of bubbles joined by data flows in the system.

**DFD SYMBOLS:**

In the DFD, there are four symbols

1. A square defines a source(originator) or destination of system data
2. An arrow identifies data flow. It is the pipeline through which the information flows
3. A circle or a bubble represents a process that transforms incoming data flow into outgoing data flows.
4. An open rectangle is a data store, data at rest or a temporary repository of data

Process that transforms data flow

Source or Destination of data

Data flow

Data Store

**CONSTRUCTING A DFD:**

Several rules of thumb are used in drawing DFD’S:

1. Process should be named and numbered for an easy reference. Each name should be representative of the process.
2. The direction of flow is from top to bottom and from left to right. Data traditionally flow from source to the destination although they may flow back to the source. One way to indicate this is to draw long flow line back to a source. An alternative way is to repeat the source symbol as a destination. Since it is used more than once in the DFD it is marked with a short diagonal.
3. When a process is exploded into lower level details, they are numbered.
4. The names of data stores and destinations are written in capital letters. Process and dataflow names have the first letter of each work capitalized

A DFD typically shows the minimum contents of data store. Each data store should contain all the data elements that flow in and out.

Questionnaires should contain all the data elements that flow in and out. Missing interfaces redundancies and like is then accounted for often through interviews.

**SAILENT FEATURES OF DFD’S**

1. The DFD shows flow of data, not of control loops and decision are controlled considerations do not appear on a DFD.
2. The DFD does not indicate the time factor involved in any process whether the dataflow take place daily, weekly, monthly or yearly.
3. The sequence of events is not brought out on the DFD.

**TYPES OF DATA FLOW DIAGRAMS**

1. Current Physical
2. Current Logical
3. New Logical
4. New Physical

**CURRENT PHYSICAL:**

In Current Physical DFD process label include the name of people or their positions or the names of computer systems that might provide some of the overall system-processing label includes an identification of the technology used to process the data. Similarly data flows and data stores are often labels with the names of the actual physical media on which data are stored such as file folders, computer files, business forms or computer tapes.

**CURRENT LOGICAL:**

The physical aspects at the system are removed as much as possible so that the current system is reduced to its essence to the data and the processors that transform them regardless of actual physical form.

**NEW LOGICAL**:

This is exactly like a current logical model if the user were completely happy with the user were completely happy with the functionality of the current system but had problems with how it was implemented typically through the new logical model will differ from current logical model while having additional functions, absolute function removal and inefficient flows recognized.

**NEW PHYSICAL:**

The new physical represents only the physical implementation of the new system.

**RULES GOVERNING THE DFD’S**

**PROCESS**

1. No process can have only outputs.
2. No process can have only inputs. If an object has only inputs than it must be a sink.
3. A process has a verb phrase label.

**DATA STORE**

1. Data cannot move directly from one data store to another data store, a process must move data.
2. Data cannot move directly from an outside source to a data store, a process, which receives, must move data from the source and place the data into data store
3. A data store has a noun phrase label.

**SOURCE OR SINK**

The origin and /or destination of data

1. Data cannot move direly from a source to sink it must be moved by a process
2. A source and /or sink has a noun phrase land

**DATA FLOW**

1. A Data Flow has only one direction of flow between symbols. It may flow in both directions between a process and a data store to show a read before an update. The later is usually indicated however by two separate arrows since these happen at different type.
2. A join in DFD means that exactly the same data comes from any of two or more different processes data store or sink to a common location.
3. A data flow cannot go directly back to the same process it leads. There must be at least one other process that handles the data flow produce some other data flow returns the original data into the beginning process.
4. A Data flow to a data store means update (delete or change).
5. A data Flow from a data store means retrieve or use.

A data flow has a noun phrase label more than one data flow noun phrase can appear on a single arrow as long as all of the flows on the same arrow move together as one package.

**LEVEL-1: SYSTEM LEVEL DATA FLOW**

A level-1 DFD describes the next level of details within the system, detailing the data flows between subsystems, which make up the whole.

### DFD for Administrator

**Admin authentication**

**Admin Master**

**Admin Master**

**Admin Master**

**Maintain login info**

**Admin Master**

**Check Modified Details of Admin**

**Check for login Details**

### DFD For Faculty:

staff Master

Member Authentication

staff Master

Check for the login details.

staff Master

Maintain login info

**LEVEL-2: Subsystem level DATA FLOW**

All the projects are feasible given unlimited resources and infinite time. It is both necessary and prudent to evaluate the feasibility of the project at the earliest possible time. Feasibility and the risk analysis are pertained in many ways.

**DFD for Administrator**

**Insert Staff**

**Check for Staff Info**

**Admin Master**

**Admin Master**

**Admin Master**

**Provide**

**security**

#### Admin Master

**Staff profiles view**

**DFD For Faculty**

**Insert Timetable**

**Check for Lecture info**

**Staff Master**

**Staff Master**

**Staff Master**

**Assigning time table each class**

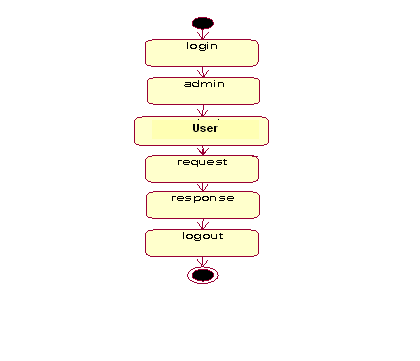
#### Staff Master

**Check for Class**

**ACTIVITY DIAGRAMS**

A State diagram/Activity diagram is a specification of the sequences of states that an object or an interaction goes through in response to events during its life, together with its responsive action. Every state diagram is having one entry and exit state. And the state can have any number of sub-states. The above state diagram represents, how admin will interact with other objects, and how he will perform actions and change his state.

**Admin Activity Diagram:**

****

**For student**

**For staff:**

login

MyProfile

View Feedback

Logout

**USE CASE DIAGRAMS**

USECASE is a description of a set of sequence of actions that a system performs that yields an observable result of value to a particular things in a model. User is an actor and these are use cases are login, view work details, assign work, approval link, view voter request details, view ward member and helper details.

**Identification of actors:**

**Actor:** Actor represents the role a user plays with respect to the system. An actor interacts with, but has no control over the use cases.

An actor is someone or something that:

* Interacts with or uses the system.
* Provides input to and receives information from the system.
* Is external to the system and has no control over the use cases.

Actors are discovered by examining:

* Who directly uses the system?
* Who is responsible for maintaining the system?
* External hardware used by the system.
* Other systems that need to interact with the system.

**Questions to identify actors:**

* Who is using the system? Or, who is affected by the system? Or, which groups need help from the system to perform a task?
* Who affects the system? Or, which user groups are needed by the system to perform its functions? These functions can be both main functions and secondary functions such as administration.
* Which external hardware or systems (if any) use the system to perform tasks?
* What problems does this application solve (that is, for whom)?
* And, finally, how do users use the system (use case)? What are they doing with the system?

**Identification of use cases:**

**Use case:** A use case can be described as a specific way of using the system from a user’s (actor’s) perspective.

**Admin Use case Diagram**

login

Add Faculty

Add student

View student faculty

View feedback

admin

logout

**SEQUENCE DIAGRAMS**

A sequence diagram is a graphical view of a scenario that shows object interaction in a time-based sequence what happens first, what happens next. Sequence diagrams establish the roles of objects and help provide essential information to determine class responsibilities and interfaces.

There are two main differences between sequence and collaboration diagrams: sequence diagrams show time-based object interaction while collaboration diagrams show how objects associate with each other.

**Object:** An object has state, behavior, and identity. The structure and behavior of similar objects are defined in their common class. Each object in a diagram indicates some instance of a class. An object that is not named is referred to as a class instance. The object icon is similar to a class icon except that the name is underlined. An object's concurrency is defined by the concurrency of its class.

**Message:** A message is the communication carried between two objects that trigger an event. A message carries information from the source focus of control to the destination focus of control. The synchronization of a message can be modified through the message specification. Synchronization means a message where the sending object pauses to wait for results.

**Link:** A link should exist between two objects, including class utilities, only if there is a relationship between their corresponding classes. The link is depicted as a straight line between objects or objects and class instances in a collaboration diagram. If an object links to itself, use the loop version of the icon.

**For admin**

****

**For Lecturer**

****

**For student:**

****

**CLASS DIAGRAM**

**Identification of analysis classes:**

A class is a set of objects that share a common structure and common behavior (the same attributes, operations, relationships and semantics). A class is an abstraction of real-world items.

There are 4 approaches for identifying classes:

1. Noun phrase approach:
2. Common class pattern approach.
3. Use case Driven Sequence or Collaboration approach.
4. Classes , Responsibilities and collaborators Approach
5. **Noun Phrase Approach:**

The guidelines for identifying the classes:

* 1. Look for nouns and noun phrases in the use cases.
  2. Some classes are implicit or taken from general knowledge.
  3. All classes must make sense in the application domain; Avoid computer implementation classes – defer them to the design stage.
  4. Carefully choose and define the class names.

After identifying the classes we have to eliminate the following types of classes:

1. Redundant classes.
2. Adjective classes.
3. **Common class pattern approach:**

The following are the patterns for finding the candidate classes:

* 1. Concept class.
  2. Events class.
  3. Organization class
  4. Peoples class
  5. Places class
  6. Tangible things and devices class.

1. **Use case driven approach:**

We have to draw the sequence diagram or collaboration diagram. If there is need for some classes to represent some functionality then add new classes which perform those functionalities.

1. **CRC approach:**

The process consists of the following steps:

* 1. Identify classes’ responsibilities ( and identify the classes )
  2. Assign the responsibilities
  3. Identify the collaborators.

**Identification of responsibilities of each class:**

The questions that should be answered to identify the attributes and methods of a class respectively are:

1. What information about an object should we keep track of?
2. What services must a class provide?

**Identification of relationships among the classes:**

Three types of relationships among the objects are:

**Association:** How objects are associated?

**Super-sub structure:** How are objects organized into super classes and sub classes?

**Aggregation:** What is the composition of the complex classes?

**Guidelines for identifying the tentative associations:**

* A dependency between two or more classes may be an association. Association often corresponds to a verb or prepositional phrase.
* A reference from one class to another is an association. Some associations are implicit or taken from general knowledge.

**Super-sub class relationships**

Super-sub class hierarchy is a relationship between classes where one class is the parent class of another class (derived class).This is based on inheritance. This hierarchy is represented with Generalization.

**Guidelines for identifying the super-sub relationship, a generalization are**

1***.* Top-down*:*** Look for noun phrases composed of various adjectives in a class name. Avoid excessive refinement. Specialize only when the sub classes have significant behavior.

2.**Bottom-up*:*** Look for classes with similar attributes or methods. Group them by moving the common attributes and methods to an abstract class. You may have to alter the definitions a bit.

3.**Reusability*:*** Move the attributes and methods as high as possible in the hierarchy.

4**. Multiple inheritances*:*** Avoid excessive use of multiple inheritances. One way of getting benefits of multiple inheritances is to inherit from the most appropriate class and add an object of another class as an attribute

`The class diagram is core to object-oriented design.  It describes the types of objects in the system and the static relationships between them.

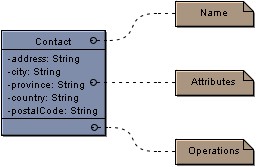
**Packages**

Packages allow you to break up a large number of objects into related groupings.  In many object oriented languages (such as Java), packages are used to provide scope and division to classes and interfaces.  In the UML, packages serve a similar, but broader purpose.

Package

**Classes**

The core element of the class diagram is the class.  In an object oriented system, classes are used to represent entities within the system; entities that often relate to real world objects.



The *Contact* class above is an example of a simple class that stores location information.

Classes are divided into three sections:

**Top**: The **name**, **package** and **stereotype** are shown in the upper section of the class

**Centre**: The centre section contains the attributes of the class.

**Bottom**: In the lower section are the **operations** that can be performed on the class.

**Attributes**

An **attribute** is a property of a class.  In the example above, we are told that a *Contact* has an address, a city, a province, a country and a postal code.  It is generally understood that when implementing the class, functionality is provided to set and retrieve the information stored in attributes. The format for attributes is:

*Visibility name: type = default Value*

The visibility is as follows:

|  |  |
| --- | --- |
| **-** | Private |
| **+** | Public |
| **#** | Protected |
| **~** | Package |

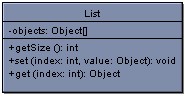
In object oriented design, it is generally preferred to keep most attributes private.

**Static**: attributes that are static only exist once for all instances of the class.  In the example above, if we set *city* to be static, any time we used the *Contact* class the *city* attribute would always have the same value.

**Final:** if an attribute is declared final, it's value cannot be changed.  The attribute is a constant.

**Operations**

The **operations** listed in a class represent the functions or tasks that can be performed on the data in the class.



In the *List* class above, there is one attribute (a private array of Objects) and three operations.

The format for operations is:

*visibility name (parameters): type*

The format is very similar to that of the attribute except with the removal of a default value and the addition of parameters.

Parameters take the format:

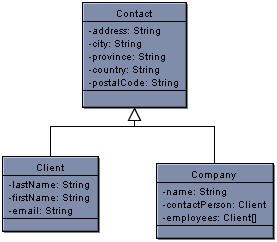
*direction name: type = default value*

The direction can be one of *in*, *out*, *input* or it can be unspecified.

In Visual Case you can show and hide the parameter list for a class or all classes on a diagram.  If the list is hidden and an operation has parameters, three dots are shown (...) to indicate that parameters exist, but are hidden.  Sometimes operations have numerous parameters that need not be shown all the time.

**Generalization**

The **generalization** link is used between two classes to show that a class incorporates all of the attributes and operations of another, but adds to them in some way.



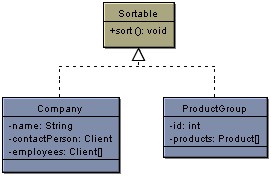
In the above diagram, we again see our *Contact* class, only now with two child classes.  We can say that *Client* and *Company* **inherit**, **generalize** or **extend** *Contact*.  In each of *Client* and *Company* all of the attributes in *Contact* (address, city, etc.) exist, but with more information added.  In the above situation *Contact* is said to be the **super class** of *Client* and *Company*.

Above, *OntarioTaxCalculator* redefines or **overrides** the implementation of the method in BasicTaxCalculator.  Essentially, the code is different but the operation is called the same way.

Sometimes you may want to force children to override methods in a parent class.  In this case you can define the methods in the super class as **abstract**.  If a class has abstract operations, the class itself is considered abstract.  Abstract methods and classes are shown in italics.  Not all of the operations in an abstract class have to be abstract.

**Interfaces**

Many object oriented programming languages do not allow for multiple inheritance.  The **interface** is used to solve the limitations posed by this.  For example, in the earlier class diagram *Client* and *Company* both generalize *Contact* but one or the other child classes may have something in common with a third class that we do not want to duplicate in multiple classes.



The interface *Sort able*, is used in the above example to show that both *Company* and *Product* implement the *sort* operation.  We can say that *Company* and *Product* **implement** *Sort able* or that they are *Sort able*.  Because Product already generalizes *Contact*, we could not also allow it to generalize *Sort able*.  Instead, we made *Sort able* an interface and added a **realization** link to show the implementation.

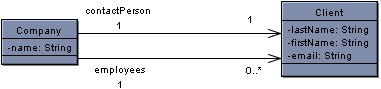
Interfaces are very similar to abstract classes with the exception that they do not have any attributes.  As well, unlike a class, all of the operations in an interface have no implementation.  The child Classes *Company* and *Product* are forced to implement the *sort* operation in its entirety.

**Associations**

Classes can also contain references to each other.  The *Company* class has two attributes that reference the Client class.



Although this is perfectly correct, it is sometimes more expressive to show the attributes as **associations**.



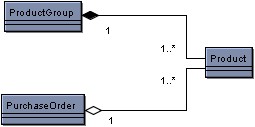
The above two associations have the same meaning as the attributes in the old version of the *Contact* class.

The first association (the top one) represents the old *contact Person* attribute.  There is one contact person in a single *Company*.  The **multiplicity** of the association is one to one meaning that for every *Company* there is one and only one *contact Person* and for each *contact Person* there is one *Company*.  In the bottom association there are zero for each company.  Multiplicities can be anything you specify.  Some examples are shown:

|  |  |
| --- | --- |
| **0** | Zero |
| **1** | One |
| **1..\*** | one or many |
| **1..2, 10..\*** | one, two or ten and above but **not** three through nine |

The arrows at the end of the associations represent their **navigability.**  In the above examples, the *Company* references *Clients*, but the *Client* class does not have any knowledge of the *Company*.  You can set the navigability on either, neither or both ends of your associations.  If there is no navigability shown then the navigability is unspecified.

**Aggregation and Composition**



The above example shows an **aggregation** association and a **composition** association.

The **composition** association is represented by the solid diamond.  It is said that *Product Group* is **composed** of *Products*.  This means that if a *Product Group* is destroyed, the *Products* within the group are destroyed as well.

The **aggregation** association is represented by the hollow diamond.  *Purchase Order* is an **aggregate** of *Products*.  If a *Purchase Order* is destroyed, the *Products* still exist.

If you have trouble remembering the difference between composition and aggregation, just think of the alphabet.  Composition means destroy and the letters 'c' and 'd' are next to each other.

**Dependencies**

A **dependency** exists between two elements if changes to one will affect the other.  If for example, a class calls an operation in another class, then a dependency exists between the two.  If you change the operation, than the dependent class will have to change as well.  When designing your system, the goal is to minimize dependencies.

PackageDiagram

To help clarify the dependencies in your design, you may wish to draw a **Package Diagram**.  A package diagram is essentially a class diagram with only packages and dependencies showing.  Dependencies can exist between any components in the UML however at the highest level, dependencies will exist between packages.  Within a package, the dependencies may be too numerous to specify.  That is not to say that numerous dependencies are okay.  Even within a package you want to limit the dependencies, however between packages in particular you should be strict about the number of dependencies that exist.  In general, the fewer the dependencies the more **scaleable** and **maintainable** your system will be

Administrator



UserName : Varchar2



Password : Varchar2



Staff()



Lecturer()



Student()

Faculty



User Name : varchar



Password : varchar



Lecturer class assign()



Student Info()



Marks Entry()



Time table Entry()



Holidays()



Student



Username : varchar



Password : varchar



Mark reports()



View schedules()



View class Tme tables()



Staff assigning Info()



Profile()

Dependency

Dependency

**COLLOBORATION DIAGRAMS**

Collaboration diagram is an interaction diagram or communication diagram. A Communication diagram models the interactions between objects or parts in terms of sequenced messages. Communication diagrams represent a combination of information taken from class, sequence and [Use Case Diagrams](http://en.wikipedia.org/wiki/Use_case_diagram) describing both the static structure and dynamic behavior of a system.

Communication diagrams use the free-form arrangement of objects and links as used in Object diagrams. In order to maintain the ordering of messages in such a free-form diagram, messages

Are labeled with a chronological number and placed near the link the message is sent over. Reading a communication diagram involves starting at message 1.0, and following the messages from object to object.

Communication diagrams show a lot of the same information as sequence diagrams, but because of how the information is presented, some of it is easier to find in one diagram than the other. Communication diagrams show which elements each one interacts with better, but sequence diagrams show the order in which the interactions take place more clearly.

### Basic Collaboration Diagram Symbols and Notations

##### Class roles

Class roles describe how objects behave. Use the UML object symbol to illustrate class roles, but don't list object attributes.

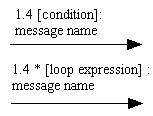
Class roles

##### Association roles

Association roles describe how an association will behave given a particular situation. You can draw association roles using simple lines labeled with stereotypes.  
Association roles

##### Messages

Unlike sequence diagrams, collaboration diagrams do not have an explicit way to denote time and instead number messages in order of execution. Sequence numbering can become nested using the Dewey decimal system. For example, nested messages under the first message are labeled 1.1, 1.2, 1.3, and so on



Admin



**For student:**

****