OOT assignment

(6) Differentiate between synchronous and asynchronous messages in terms of object oriented modelling.

Ans) Synchronous messages:

- It includes a client that waits for the server to respond to a message. Messages are able to flow in both directions. It is a 2-way communication.
- If a caller sends a synchronous message it must wait until the message is done, such as invoking a subroutine.

Asynchronous Message:

- It involves a client that does not wait for a message from the server. An event is used to trigger a message from a server, it is a 1-way communication.
- If a caller sends an asynchronous message, it can continue processing & doesn't have to wait for a response.
- Asynchronous caller is a multithread application & in messages oriented middleware. Asynchming gives better responsiveness & reduces the temporal coupling but is harder to debug.

Que → What are the steps for designing an object?

Ans) Object design includes the following phases:

(1) Object identification - The objects identified in the object oriented analysis phase are grouped into

classes & refined so that they are suitable for actual implementation functions of the stage are:
- identifying & refining the classes in each subsystem or package.
- defining the links & association between the classes.
- Designing aggregations
- Designing the hierarchial association among the classes, i.e., specialization/generalization and inheritance.

② Object Representation:
Once the classes are identified, they need to be represented using object modelling techniques. This stage essentially involves constructing UML diagrams. There are 2 types of models that need to be produced:
- Static models- To describe the static structure of the system using class structure of a system using class diagram & object diagram.
- Dynamic models- To describe the dynamic structure of a system & show the interaction b/w classes using interaction diagram & state-chart diagram.

③ Classification of Operation -
In this the operation to be performed on objects are defined by combining the 3 models developed in the OOA phase, namely, object model, dynamic model and function model. An operation specifies what is to be done not how it is to be done.

④ Algorithmic Design -
The operations in the objects are defined using algorithms. An algorithm is a stepwise procedure

that solves the problem laid down in an operation. It focuses on how it is to be done. There may be more than one algorithm corresponding to an operation. The metrics for choosing optimal algorithm are-
- Computational complexity
- Flexibility
- Understandability

(5) Design of Relationships:-
- The main relationships that are addressed comprises of associations, aggregation and inheritance.

The designer should do the following regarding associations:
→ Identify whether an association is unidirectional or bidirectional.
→ Analyze path of association & update them if necessary.

Regarding inheritance, the designer should do-
- Adjust the classes & their associations.
- Identify the abstract classes.
- Make provision so that behaviours are shared when needed.

(6) Implementation of Control:
- Represent state at a location within a program
- State Machine engine.
- Control as concurrent tasks.

(7) Packaging Classes:
During object design, classes & objects are grouped into package to enable, multiple groups to work cooperati on a project. Different aspects of packging are-
- Hiding internal information from outside view.
- Coherence of elements.
- construction of physical modules.

Non-Object-Oriented Languages

* Mapping Object-Oriented Concepts

Implementing an object-oriented design in a non-object oriented language requires basically the same steps as implementing a design in an object oriented language. The programmer using a non-object oriented language must map object-oriented concepts into the target language, whereas the compiler for an object-oriented language performs such a mapping automatically. The steps required to implement a design are:

1) Translate classes into a data structures
2) Pass arguments to method.
3) Allocate storage for objects.
4) Implement inheritance in data structures.
5) Implement method resolution. *
6) Implement associations.
7) Deal with concurrency. *
8) Encapsulate internal details of classes.

Implementation in C

The language C, with its loose type checking provides the flexibility of that provides the flexibility that allows several important object-oriented concepts to be implemented. The C-pointer mechanism and run time memory allocation also assist the implementation. Implement classes, instances, single inheritance and run time method resolution in C with little loss of efficiency is quite simple.

1) TRANSLATING CLASSES INTO DATA STRUCTURES

Normally you will implement each class as a single contiguous block of attributes - a record structure. Each attribute in a class becomes an element in the record, has a declared type which can be a primitive type, such as integer, real or character or can be a structured value, such as an embedded record structure or a fixed-length array. A variable that identifies an object must therefore be implemented as a shareable reference and not simply by copying the values of an object's attributes.

Translating classes into C struct Declarations

Each class in the design becomes a C struct. Each attribute defined in the class becomes a field of the C struct. Structured for the Window class declared as:

```
struct Window
{
  Length xmin;
  Length ymin;
  Length xmax;
  Length ymax;
};
```

Length is a C type (not a class) defined with a C typedef statement to provide greater modularity in the definition of values:

```
typedef float Length;
```

In C, an object reference can be represented by a pointer to its object record:

```
Struct Window * window;
Length x1 = window->xmin;
```

An object can be allocated statically, automatically (on the stack), or dynamically (on the heap) because C can compose a pointer to any object, including one embedded within another structure.

2) Passing Arguments to Methods

Every method has at least one argument. In a non-object-oriented language, the argument must be made explicit. of course. Some of the arguments may be simple data values and not objects.

In passing an object as an argument to a method, a reference to the object must be passed if the value of the object can be updated within the method. Passing all arguments as references is more uniform, however.

A consistent naming convention for method function names. One that we have found useful in C is to concatenate the class name, two underscores, and the operation name. (The two underscores separate the class name from the operation name, each of which may contain single underscores.)

Passing arguments in C

In C, an object should always be passed by pointer. Although C permits structures to be passed by value, passing a pointer to an object structure is usually more efficient and provides a uniform access mechanism for both query and update operations.

```
Window__add_to_selections (self, shape)
    Struct Window * self;
    Struct Shape * shape;
```

3) ALLOCATING OBJECTS :- Objects can be allocated statically (at compile time), dynamically (from a heap), or on a stack. Statically allocated objects are implemented as global variables allocated by the compiler. Their lifetime is the duration of the program.

Most temporary and intermediate objects will be implemented as stack-based variables (such as C automatic variables or Pascal local variables).

The advantage of stack-based variables is that they are automatically allocated and deallocated. The programmer must ensure that no references to a stack-based object remain after the declaring block has been exited.

- Dynamically allocated objects are needed when the no. of them is not known at compile time. A general object can be implemented as a data structure allocated on request at run time from a heap. Storage for dynamically allocated objects is requested explicitly by a special operator: malloc in C, new in Pascal or Ada, make in CommonLisp.
- Once allocated, dynamic objects persist until they are explicitly deallocated, so pointers to them can be stored in other objects
- Some languages, such as Lisp, provide garbage collection which removes the burden of deallocation from the programmer and avoids the danger of dangling pointers.

Allocating objects in C :- Global objects can be declared as top-level struct variables. They can be initialize at compile time:

```
struct Window outer_window = {0.0, 0.0, 8.5, 11.0};
```

When calling a method, the address of the variable (&outer_window) must be passed. Most object should be allocated dynamically using malloc or calloc:

```
struct Window* create_window (xmin, ymin, width, height)
    Length xmin, ymin, width, height;
{
    struct Window *window;
    window=(struct Window *) malloc (sizeof (struct Window
```

```
window -> xmin = xmin;
window -> ymin = ymin;
window -> xmax = xmin + width;
window -> ymax = ymin + height;
return window;
}
```

When an object is no longer needed, it must be deallocated using the C free function. Make sure that there are no pointers to it before deallocating it. Also be sure to free any component objects that are pointed to by its instance variables.

- Temporary and intermediate objects can be allocated as ordinary C automatic variables within a function body or block.
- When calling a method, the address of the variable must be passed.
- Stack based variables cannot be used when their lifetime outlasts the function that created them.

4) IMPLEMENTING INHERITANCE: There are several ways to implement data structures for inheritance in a non-object-oriented language:

- Avoid it → Many applications do not require inheritance
  - Many other applications have only few classes requiring inheritance.
- Flatten the class hierarchy → Use inheritance during design but expand each concrete class as an independent data structure during implementation.

→ Each inherited operation must be reimplemented as a separate method on each concrete class

→ One useful technique is to group some inherited attributes into a record type and embed the record

in each concrete class to reduce the no. of duplicated lines in each declaration.

- Break out separate objects → Instead of inheriting common attributes from a superclass, a group of attributes can be pulled out of all the subclasses and implemented as a separate objects with a reference to it stored within each subclass.

Implementing inheritance in C → To handle single inheritance, embed the declaration for the superclass as the first part of each subclass declaration.

- The first field of each struct is a pointer to a class descriptor object shared by all direct instance of a class.
- The class descriptor object is a struct containing the class attributes, including name of the class and the methods for the class
- The class shape is an abstract class, with concrete subclasses Box and Circle. The C declarations for classes Shape, Box and Circle are as follows:

```
Struct Shape
{
  Struct ShapeClass * class;
  Length x;
  Length y;
};
Struct Box
{
  Struct Boxclass * class;
  Length x;
  Length y;
  Length width;
  Length height;
};
Struct Circle
{
```

```
Struct CircleClass * class;
Length x;
Length y;
Length radius;
};
```

- A pointer to a Box or Circle structure can be passed to a C function expecting a pointer to a shape structure because the first part of the Box or circle structure is identical to the shape structure.
- The prefix of the structure is the same as the superclass structure, the superclass method simply ignores the extra field on the end

for example. a pointer to Box is interpreted as a pointer to shape in the following call:

```
Struct Box * box;
Struct Window * window;
Window_add_to_selections (window, box)
```

The first field of each structure is a pointer to the class descriptor for the actual class of each object instance.

5) IMPLEMENTING ~~METHOD RESOLUTION~~ ASSOCIATIONS

Implementing association in a non-object-oriented language presents the same two possibilities as in an O-O language: mapping them into pointers or implementing directly as association container objects.

- Mapping associations to pointers. The traditional approach to implementing binary associations is to map each role of an association into an object pointer stored as a field of the source object record.

- Each object contains a pointer to an associated object or a pointer to a set of associated objects (if the multiplicity greater than one).
- The association may be implemented in one direction or in both direction. if it is traversed in only a single direction
- It may be implemented ~~in one direction~~ as a pointer from one object to another. if it traversed in both directions, it must be implemented using pointers in both associated objects; the two pointers must be kept mutually consistent.

• Implementing association objects. An association can be implemented directly as a data structure. if an association relates more than two classes then it be mapped into pointer and a separate object

- An association is simply a set of records, each containing one object ID for each associated classes.
- The simplest approach is to implement an association as an array or list of records or record pointers.
- A more efficient implementation is to sort the list on one or more key fields or to hash the elements on one or more key fields. Such a data structured is said to be indexed on the key fields.
- Whenever an element is added, modified or deleted, all the indexes must be updated accordingly. depending upon the relative no.s of updates and accesses in each direction of traversal.

Implementing Associations in C - A binary association is usually implemented as a field in one each associated object, containing a pointer to the related object or to an array of related objects for example - The many to one association b/w item and group will be implemented as:-

```
Struct Item
{
  Struct ItemClass * class;
  Struct Group * group;
  Struct Group
  {
    Struct GroupClass * class;
    int item_count;
    Struct Item ** items;
  };
```

other data structures, such as a linked list or a hash table, can also be used to store sets of objects. In this example a group is created from a set of selected items.

- The memory for the items pointer can be allocated all at once, since the number of items in a group does not change.
- if it were possible to add a single new item to a group, then both pointers must be updated.

```
Group_add_item (self, item)
    Struct Group * self;
    Struct Item * item;
{ item->group = self;
  self->items = (struct Item**) realloc (self->items
                  ++self->item_count * sizeof (struct Item*));
  self->items [self->item_count-1] = item;
}
```

6) **ENCAPSULATION** → Encapsulation of data representation and method implementation is one of the major themes of Object-oriented programming. Object-Oriented languages provide constructs to encapsulate implementation.

- Some of the encapsulation is lost when the programmer must manually translate object-oriented concepts into a non-object-oriented language. but you can still take advantage of the encapsulation facilities provided by the language.

Encapsulation in C → C has a reputation for encouraging loose programming style harmful to encapsulation.

→ Steps to improve encapsulation.

- Avoid the use of global variables.
- Package the methods for each class into a separate file. Only include declarations for classes whose internal structure you need, such as ancestor classes of the current class.
- Do not access the fields of objects of different classes; call an access method instead.
- Treat objects of other classes as type "void *"; while technically illegal, most compilers treat all pointers the same.

Procedure Oriented Programming

1. In POP, program is divided into small parts called functions.
2. POP does not have any access specifier.
3. In POP, Data can move freely from function to function in the system.
4. POP does not have any proper way for hiding data, so it is less secure.
5. In POP, Most function uses global data for sharing that can be accessed freely from function to function in the system.
6. POP follows Top Down approach
7. In POP, Importance is not given to data but to functions as well as sequence of actions to be done.
8. In POP, Overloading is not possible.
9. Example:- C, VB, FORTRAN, Pascal.

Object oriented Programming

1. program is divided into parts called objects.
2. OOP has access specifier named Public, Private, Protected etc.
3. In OOP, objects can move & communicate with each other through member functions.
4. OOP provides Data Hiding so provides more security.
5. In OOP, data can't move easily from function to function, it can be kept public or private so we can control the access of data.
6. OOP follows Bottom Up approach.
7. In OOP, Importance is given to the data rather than procedures or functions because it works as a real world.
8. In OOP, overloading is possible in the form of function Overloading & Operator Overloading.
9. C++, JAVA, VB.NET, C#.NET.

Encapsulation in C++ :- Encapsulation is a process of combining data members & functions in a single unit called class. It is one of the popular feature of OOPs that helps in data hiding.

Example :- Step - ① Make all the data member private
② Create public setter & getter functions for each data member in such a way that the set function set the value of data member & get function get the value of data member.

```
※ include < iostream>
   using namespace std;
   class Example Encap{
Private :
     int num;
     char ch;
public :
   int getNum( ) const {
        return num;
   }
   char getch( ) const {
     return ch;
   }
   void set Num (int num). {
        this -> num = num;
   }
   void set ch (char ch) {
      this -> ch = ch;
   }
};
int main ( ) {
   Example Encap obj;
   obj. set Num(100);
   obj. set Ch ('A');
   cout << obj. get Num ( ) << end l;
   cout << obj. getCh ( ) << end l;
   return 0;
```

output :
100
A

Abstraction ~~is one of~~ in C++ :- Abstraction is one of the feature of OOP, where you show only relevant details to the user & hide irrelevent details.

Example :-

```
#include <iostream>
using namespace std;
class AbstractionExample {
private:
    int num;
    char ch;
public:
    void setMyvalues (int n, char c) {
      num = n;  ch = c;
    }
    void getMyValues() {
      cout << "Number is: " << num << endl;
      cout << Char is : " << ch << endl;
    }
};
int main() {
    AbstractionExample obj;
    obj.setMyvalues (100, 'X');
    obj.getMyvalues();
    return 0;
}
```

Output

Number is : 100

Char is : X

Comparison of Methodologies →

In this chapter, the comparison is made between different software engineering approaches and OMT (Object Modeling Technique).

The aim is to identify the major differences and similarities between the OMT and other approaches

The software approaches discussed in this chapter are—

i) SA/SD (Structured Analysis/ Structured Design)
ii) JSD (Jackson structured Development)

we will discuss these methodologies and compare them with OMT.

(i) SA/SD (Structured Analysis / Structured Design)

This methodology works on data flow diagram approach. In other terms SA/SD is the representative of the data flow approach.
This approach consists of two phases—

(i) Analysis Phase.
(ii) Design Phase.

During the analysis phase, data flow diagrams, process specifications, data dictionary, state transition diagrams and entity-relationship diagrams are used to logically describe a system

In the design phase, details are added to the analysis models and data flow diagrams are converted into structure chart descriptions of programming langauge

Data Flow Diagrams are model the transformation of data and are the focus of SA/SD. It consists of processes, data flows, actors and ~~flow diagrams~~ data stores.

SA/SD recursively divides complex processes into subdiagrams, until many small processes are left that are easy to implement.

the decomposition steps when the resulting processes are simple enough and process specification is written for each lowest level process.

Comparison with OMT⇒

Both the models use similar modeling constructs and supports the three orthogonal views of a system.

the difference between these two models is primarily a matter of style and emphasis.

① In SD/SA model

Functional model > Dynamic model > Object Model (F > D > O) according to importance.

In OMT model

(O > D > F).

② SA/SD ~~is~~ organises a system around procedures while (OMT) organises a system around real world objects or conceptual objects

3) In SA/SD, the decomposition of a process into subprocesses is somewhat arbitary while in OMT decompositions is based on objects in the problem domain.

ii) JSD (Jackson Structured Development)

JSD is a different methodology than SA/SD or OMT. It is a mature methodology.
It doesn't distinguish between analysis and design. Instead, lumps both phases together as specification.

JSD divides the system development into two stages →

1) Specification
2) Implementation.

JSD first determines the "what" and then how. JSD uses graphic models.

JSD model begins with consideration of the real world.

It describes the real world in terms of entities, actions and ordering of actions.

JSD software development consists of six sequential steps —

B) Entity action →

In this step, the developer lists entities and actions for part of the real world.
The input to the entity action step is the requirement statement and output is a list of entities and actions

eg → design of an elevator control system
This system controls two elevators which service six floors. Each elevator has six inside buttons one for each floor. Each floor has up and down button. Two entities are defined: button and elevator. Three actions are identified as:
Press a button, elevator arrives at floor n and elevator leaves floor n.

entity structure step partially orders the action of each entity by time.
arrive and leave actions must go on alternatively
Arriving at two different two different floors can't be in successions..

initial model step states how the real world connects to the abstract model.
JSD supports state vector system and data stream connection.

Illustration of state vector connection:

When the user presses the button for more than one time. He/she doesn't want the system to recognize the number of times again and again. The up button's pressing sets the "up flag" to true. i.e. it has no further effect that how many times the user press the button. JSD model is unaware of the number of presses and only communicates with the real world through "up-flag". Jackson calls this "up-flag" a state-vector connection.

The function step uses pseudocode to state outputs of actions. At the end of this the developer has a complete specification of the required system. In the elevator example, turning the display panel lights on and off as an elevator arrives at each floor is a function that must be specified.

The system timing step considers how much the model is permitted to lag the real world. For eg. an elevator control system must detect when up and down button is pressed.

The implementation step focuses on the problems of process scheduling and allocate the ~~processes to the processors~~. processors to the

After the six JSD steps comes writing of code and database design.

Comparison with OMT →

JSD approach more difficult because of reliance on pseudocode: graphic models are easier to understand. JSD is also complex but it was designed for heavy, real based problems. JSD's complexity is unnecessary for the more common, simpler problems.

Jackson places more emphasis on actions and less on attributes than we do.

Some JSD looks like OMT associations too. Eg: a clerk allocates a product to an order. We call allocates an association, Jackson calls it an action.

Jackson finds attributes confusing and gives stress on actions more

JSD is useful methodology (applications)

1) Concurrent software.
2) Real time software.
3] Microcode
4] Programming Parallel computers

JSD is ill suited / non useful

1) High Level Analysis
2] Databases
3] Conventional software

ABES Engineering College
College Code-032

Object Oriented Analysis and Object oriented design

UNIT-III

OBJECT ORIENTED SYSTEM DESIGN(OOSD)

Batch 2023-2024

Overview of object design

- Object-oriented analysis, design and programming are related but distinct.
- OOA is concerned with developing an object model of the application domain.
- OOD is concerned with developing an object-oriented system model to implement requirements.
- OOP is concerned with realizing an OOD using an OO programming language such as Java or C++.

Characteristics of OOD

- Objects are abstractions of real-world or system entities and manage themselves.
- Objects are independent and encapsulate state and representation information.
- System functionality is expressed in terms of object services.
- Shared data areas are eliminated.
- Objects communicate by message passing.
- Objects may be distributed and may execute sequentially or in parallel.

Advantages of OOD

- Easier maintenance.
- Objects may be understood as stand-alone entities.
- Objects are potentially reusable components.
- For some systems, there may be an obvious mapping from real world entities to system objects.

Object Design

- The object design phase determines the full definitions of the classes and associations used in the implementation, as well as the interfaces and algorithms of the methods used to implement operations.
- The object design phase adds internal objects for implementation and optimizes data structures and algorithms.
- The objects discovered during analysis serve as the skeleton of the design, but the object designer must choose among different ways to implement them with an eye toward minimizing execution time, memory and other measures of cost.
- The operations identified during the analysis must be expressed as algorithms, with complex operations decomposed into simpler internal operations.
- Optimization of the design should not be carried to excess, as ease of implementation, maintainability, and extensibility are also important concerns.

Steps of Design

During object design, the designer must perform the following steps:

1. Combining the three models to obtain operations on classes.
2. Design algorithms to implement operations.
3. Optimize access paths to data.
4. Implement control for external interactions
5. Adjust class structure to increase inheritance.
6. Design associations.
7. Determine object representation.
8. Package classes and associations into modules.

Combining the three models to obtain operations on classes.

- After analysis, we have ***object, dynamic and functional model***, but the object model is the main framework around which the design is constructed.
- **Object Model:** An object model consists of the following important features:

a. Object Reference

b. Interfaces

c. Actions

d. Exceptions

e. Garbage collection

- **Dynamic Model:** The dynamic model is used to express and model the behavior of the system over time. It includes support for activity diagrams, state diagrams, sequence diagrams and extensions including business process modeling.
- **Functional Model:** A function model or functional model in systems engineering and software engineering is a structured representation of the functions (activities, actions, processes, operations) within the modeled system or subject area.

  A function model, also called an activity model or process model, is a graphical representation of an enterprise's function within a defined scope. The purposes of the function model are to describe the functions and processes, assist with discovery of information needs, help identify opportunities, and establish a basis for determining product and service costs.

- The object model from analysis may not show operations. The designer must convert the actions and activities of the dynamic model and the processes of the functional model into operations attached to classes in the object model.
- Each state diagram describes the life history of an object. A transition is a change of state of the object and maps into an operation on the object. In the state diagram, the action performed by a transition depends on both the event and the state of the object. Therefore, the algorithm implementing an operation depends on the state of the object.
- An event sent by an object may represent an operation on another object .Events often occur in pairs , with the first event triggering an action and the second event returning the result on indicating the completion of the action.

- An action or activity initiated by a transition in a state diagram may expand into an entire DFD in the functional model .
- The network of processes within the DFD represents the body of an operation. The flows in the diagram are intermediate values in operation.
- The designer convert the graphic structure of the diagram into linear sequence of steps in the algorithm .
- The process in the DFD represent sub-operations. Some of them, but not necessarily all may be operations on the original target object or on other objects.

Designing algorithms

- Each operation specified in the functional model must be formulated as an algorithm. The analysis specification tells what the operation does from the view point of its clients, but the algorithm shows how it is done.
- An algorithm may be subdivided into calls on simpler operations, and so on recursively, until the lowest-level operations are simple enough to implement directly without refinement .
- The algorithm designer must decide on the following:

i. Choosing algorithms

ii. Choosing Data Structures

iii. Defining Internal Classes and Operations

iv. Assigning Responsibility for Operations

i. Choosing algorithms

Considerations in choosing among alternative algorithm include:

a) **Computational Complexity:** It is essential to think about complexity i.e. how the execution time (memory) grows with the number of input values.

b) **Ease of implementation and understandability:** It is worth giving up some performance on non critical operations if they can be implemented quickly with a simple algorithm.

c) **Flexibility:** Most programs will be extended sooner or later. A highly optimized algorithm often sacrifices readability and ease of change. One possibility is to provide two implementations of critical applications, a simple but inefficient algorithm that can be implemented, quickly and used to validate the system, and a complicated but efficient algorithm whose correct implementation can be checked against the simple one.

d) **Fine Timing the Object Model:** We have to think, whether there would be any alternatives, if the object model were structured differently.

ii) Choosing Data Structures

- Choosing algorithms involves choosing the data structures they work on.
- We must choose the form of data structures that will permit efficient algorithms.
- The data structures do not add information to the analysis model, but they organize it in a form convenient for the algorithms that use it.

iii) Defining Internal Classes and Operations

- During the expansion of algorithms, new classes of objects may be needed to hold intermediate results.
- New, low level operations may be invented during the decomposition of high level operations.
- A complex operation can be defined in terms of lower level operations on simpler objects.
- These lower level operations must be defined during object design because most of them are not externally visible.
- There is a need to add new internal operations as we expand high –level functions. When you reach this point during the design phase, you may have to add new classes that were not mentioned directly in the client‟s description of the problem. These low-level classes are the implementation elements out of which the application classes are built.

iv) Assigning Responsibility for Operations

- Many operations have obvious target objects, but some operations can be performed at several places in an algorithm, by one of the several places, as long as they eventually get done.
- Such operations are often part of a complex high-level operation with many consequences.
- When a class is meaningful in the real world, then the operations on it are usually clear. During implementation, internal classes are introduced.

Design Optimization

- The basic deign model uses the analysis model as the framework for implementation .
- The analysis model captures the logical information about the system, while the design model must add details to support efficient information access.
- The inefficient but semantically correct analysis model can be optimized to make the implementation more efficient, but an optimized system is more obscure and less likely to be reusable in another context.
- The designer must strike an appropriate balance between efficiency and clarity.
- During design optimization, the designer must

I. Add Redundant Associations for Efficient Access
II. Rearranging Execution Order for Efficiency
III. Saving Derived Attributes to Avoid Re-computation:

Implementation of Control

- The designer must refine the strategy for implementing the state – event models present in the dynamic model. There are three basic approaches to implementing the dynamic model:

- **State as Location within a Program:**

  This is the traditional approach to representing control within a program. The location of control within a program implicitly defines the program state. Any finite state machine can be implemented as a program. Each state transition corresponds to an input statement. After input is read, the program branches depending on the input event received. Each input statement need to handle any input value that could be received at that point.

- **State machine engine:**

The most direct approach to control is to have some way of explicitly representing and executing state machine. This approach allows you to quickly progress from analysis model to skeleton prototype of the system by defining classes from object model state machine and from dynamic model and creating ―stubs‖ of action routines. A stub is a minimal definition of function /subroutine without any internal code.

- **Control as Concurrent Tasks:**

An object can be implemented as task in programming language /operating system. It preserves inherent concurrency of real objects. Events are implemented as inter task calls using facilities of language/operating system. Concurrent C++/Concurrent Pascal support concurrency. Major Object Oriented languages do not support concurrency.

Adjustment of Inheritance

The definitions of classes and operations can often be adjusted to increase the amount of inheritance The designer should:

- **Rearrange classes and operations:** Sometimes the same operation is defined across several classes and can easily be inherited from a common ancestor, but more often operations in different classes are similar but not identical. By slightly modifying the definitions of the operations or the classes , the operations can often be made to match so that they can be covered by a single inherited operation.
- **Abstracting Out Common Behavior:** Reexamine the object model looking for commonality between classes. New classes and operations are often added during design. If a set of operations / attributes seems to be repeated in two classes , it is possible that the two classes are specialized variations of the sane thing. When common behavior has been recognized , a common super class can be created that implements the shared features , specialized features in subclass. This transformation of the object model is called abstracting out a common super class / common behavior .
- **Use Delegation to Share Implementation:** Sometimes programmers use inheritance as an implementation technique with no intention of guaranteeing the same behavior. Sometimes an existing class implements some of the behavior that we want to provide in a newly defined class, although in other respects the two classes are different. The designer may inherit from the existing class to achieve part of the implementation of the new class. This can lead to problems –unwanted behavior.

Design of Associations

During object design phase, we must formulate a strategy for implementing all associations in the object model.

- **Analyzing Association Traversal**: Associations are inherently bidirectional. If association in your application is traversed in one direction, their implementation can be simplified.
- **One-way association***:* If an association is only traversed in one direction it may be implemented as pointer.
- **Two-way associations***:* Many associations are traversed in both directions, although not usually with equal frequency.
- **Link Attributes:** Its implementation depends on multiplicity. If it is a one-one association , link attribute is stored in any one of the classes involved. If it is a many-one association, the link attribute can be stored as attributes of many object ,since each —many object appears only once in the association. If it is a many-many association, the link attribute can‟t be associated with either object; implement association as distinct class where each instance is one link and its attributes.

Object Representation

- Implementing objects is mostly straight forward, but the designer must choose when to use primitive types in representing objects and when to combine groups of related objects.
- Classes can be defined in terms of other classes, but eventually everything must be implemented in terms of built-in-primitive data types, such as integer strings, and enumerated types.
- For example, consider the implementation of a social security number within an employee object. It can be implemented as an attribute or a separate class.
- Defining a new class is more flexible but often introduces unnecessary indirection. In a similar vein, the designer must often choose whether to combine groups of related objects.

Physical Packaging

Programs are made of discrete physical units that can be edited, compiled, importe manipulated. In C and Fortran the units are source files; In object oriented languages, there are various degrees of packaging. Packaging involves the following issues:

- **Hiding internal information from outside view**: Hiding internal information permits implementation of a class to be changed without requiring any clients of the class to modify code.
- **Coherence of entities**: One important design principle is coherence of entities. An entity ,such as a class , an operation , or a module , is coherent if it is organized on a consistent plan and all its parts fit together toward a common goal.
- **Constructing physical modules:** During analysis and system design phases we partitioned the object model into modules.

Documenting Design Decisions

- The design decisions must be documented when they are made, or you will become confused. This is especially true if you are working with other developers. It is impossible to remember design details for any non trivial software system, and documentation is the best way of transmitting the design to others and recording it for reference during maintenance.
- The design document is an extension of the Requirements Analysis Document.
- The design document includes revised and much more detailed description of the object model- both graphical and textual.
- Functional model will also be extended which specifies all operation interfaces by giving their arguments, results, input-output mappings and side effects.
- From analysis to design, it is probably a good idea to keep the Design Document distinct from the Analysis Document. Because of the shift in viewpoint from an external user's view to an internal implementer's view, the design document includes many optimizations and implementation artifacts.

- **Design Document = Detailed Object Model +**
  Detailed Dynamic Model +
  Detailed Functional Model

Structured analysis and structured design (SA/SD), Jackson Structured Development (JSD)

Introduction

- Object oriented design is a method where developers think in terms of objects instead of procedures or functions.
- SA/SD approach is based on the data flow Diagram.
- JSD approach is action-oriented.
- SA/SD is easy to understand but it focuses on well defined system boundary whereas JSD approach is too complex and does not have any graphical representation.

SA/SD

- SA/SD is diagrammatic notation which is design to help people understand the system.
- The basic goal of SA/SD is to improve quality and reduce the risk of System failure.
- It establishes concrete management specification and documentation. It focuses on reliability, flexibility and maintainability of system.
- In this system it involves 2 phases.

1. Analysis Phase: It uses DFD, Data Dictionary, State Transition diagram and ER diagram.
2. Design Phase: Structure Chart, Pseudo Code

SA/SD

Analysis Phase:

- Data Flow Diagram: In a DFD model describe how the data flows through the system.
- Data Dictionary: The content that is missing from DFD is described in the data dictionary. Data Dictionary defines the Data store and there relevant meaning.
- State Transition Diagrams is similar to Dynamic model .It specifies how much time function will take to execute and data access triggered by events.
- ER Diagram: It specifies the relationship between data store

SA/SD

Design Phase: Structure Chart, Pseudo Code

- Structure Chart: It is created by the DFD. Structure Chart specifies how DFD‟s processes are grouped in to task and allocate to CPU‟s.
- Pseudo Code: Actual implementation of the System.

Jackson Structured Development (JSD)

- JSD was introduced by Michael Jackson in 1983.
- The fundamental principle of JSD is that it focuses on describing the real world by the system i.e. its main focus is to map the progress in the real world rather than specifying the functions performed by the system.
- Jackson System Development (JSD) is a method of system development that covers the software life cycle either directly or by providing a framework into which more specialized techniques can fit.
- JSD can start from the stage in a project when there is only a general statement of requirements.

Phases of (JSD)

- JSD has 3 phases:

Modeling Phase:

In the modeling phase of JSD the designer creates a collection of entity structure diagrams and identifies the entities in the system, the actions they perform, the attributes of the actions and time ordering of the actions in the life of the entities.

Specification Phase:

This phase focuses on actually what is to be done? Previous phase provides the basic for this phase. An sufficient model of a time-ordered world must itself be time-ordered. Major goal is to map progress in the real world on progress in the system that models it.

Jackson Structured Development (JSD)

Implementation Phase:

In the implementation phase JSD determines how to obtain the required functionality. Implementation way of the system is based on transformation of specification into efficient set of processes. The processes involved in it should be designed in such a manner that it would be possible to run them on available software and hardware.

JSD Steps

Initially there were six steps when it was originally presented by Jackson:

1. Entity/action step:
2. Initial model step:
3. Interactive function step
4. Information function step
5. System timing step
6. System implementation step

JSD Steps

1. **Entity/action step:** The entities perform actions in time. The action is atomic and cannot be decomposed further.
2. **Initial model step:** we concentrate on state vector and data stream communication.
3. **Interactive function step:** consist of pseudo code to the output of action generated from states. In this step the developer requires a complete specification of the system.
4. **Information function step:** consist of pseudo code to the output of action generated from states. In this step the developer requires a complete specification of the system.
5. **System timing step:** relates with the performance constraints of the system.
6. **System implementation step:** focuses on process scheduling and allocates the number of processors required. The process may be different form the processors

Jackson Structured Development (JSD)

Later some steps were combined to create method with only three steps:

1. **Modeling Step:** This Stage comprises of Entity Action and Entity Structure.
2. **Network Step:** Initial model step, Function Step, and System Timing Step.
3. **Implementation Step:** Implementation Stage

Merits of JSD

- It is designed to solve real time problem.
- JSD modeling focuses on time.
- It considers simultaneous processing and timing.
- It is a better approach for micro code application.

Demerits of JSD

- It is a poor methodology for high level analysis and data base design.
- JSD is a complex methodology due to pseudo code representation.
- It is less graphically oriented as compared to SA/SD or OMT.
- It is a bit complex and difficult to understand.

Data Flow Diagram (DFD)

Data flow diagram (DFD)

- A process model used to depict the flow of data through a system and the work or processing performed by the system. Synonyms are bubble chart, transformation graph, and process model.
- The DFD has also become a popular tool for business process redesign.
- A graphical tool, useful for communicating with users, managers, and other personnel.
- Used to perform structured analysis to determine logical requirements.
- Useful for analyzing existing as well as proposed systems.
- Focus on the movement of data between external entities and processes, and between processes and data stores.

Differences Between DFDs and Flowcharts

- Processes on DFDs can operate in parallel (at-the-same-time)

  -Processes on flowcharts execute one at a time

- DFDs show the flow of data through a system

  -Flowcharts show the flow of control (sequence and transfer of control)

- Processes on a DFD can have dramatically different timing (daily, weekly, on demand)

  -Processes on flowcharts are part of a single program with consistent timing

Why DFD ?

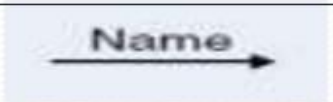
- **Provides an overview of-**
  - What data a system processes
  - What transformations are performed
  - What data are stored
  - What results are produced and where they flow

- **Graphical nature makes it a good communication tool between-**
  - User and analyst
  - Analyst and System designer

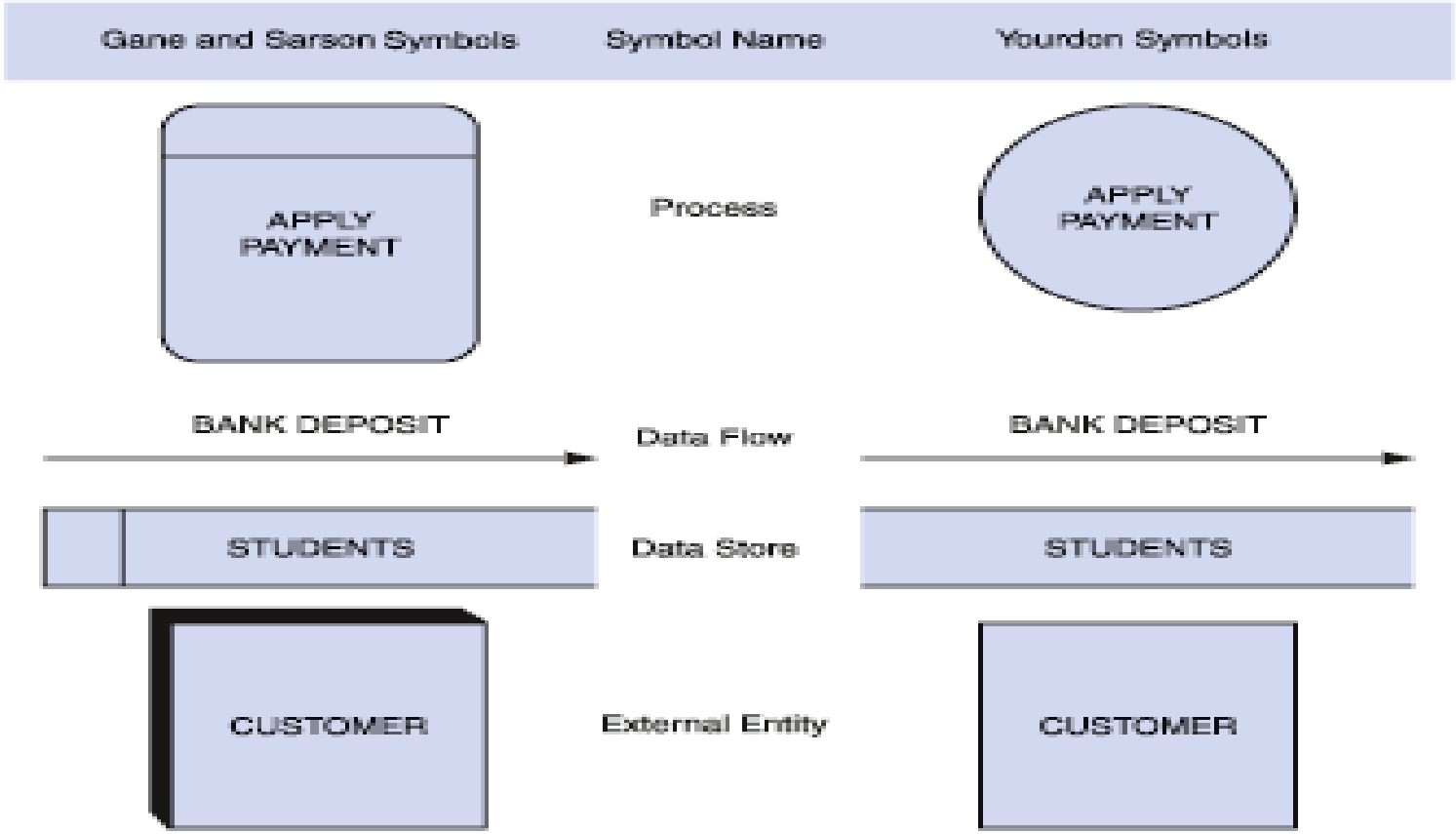
DFD elements

- Source/Sinks (External entity)
- Processes
- Data Stores
- Data flows

DFD Symbols

Symbol	Gane & Sarson Symbol	DeMarco & Yourdan Symbol
External Entity		
Process		
Data store		
Data flow		

DFD Symbols


Gane and Sarson Symbols
Symbol Name
Yourdon Symbols
APPLY PAYMENT
Process
APPLY PAYMENT
BANK DEPOSIT
Data Flow
BANK DEPOSIT
STUDENTS
Data Store
STUDENTS
CUSTOMER
External Entity
CUSTOMER

Descriptions

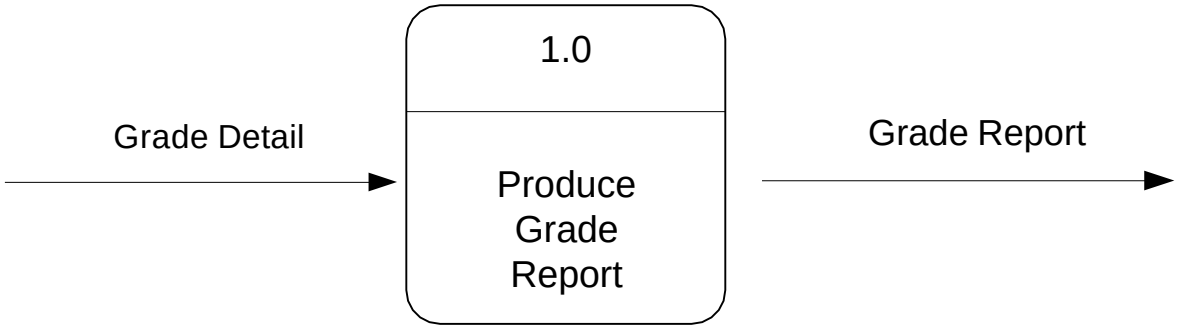
- **External Entity** - People or organizations that send data into the system or receive data from the system.
- **Process** - models what happens to the data

  i.e. transforms incoming data into outgoing data.
- **Data Store** - represents permanent data that is used by the system.
- **Data Flow** - models the actual flow of the data between the other elements.

Symbol naming

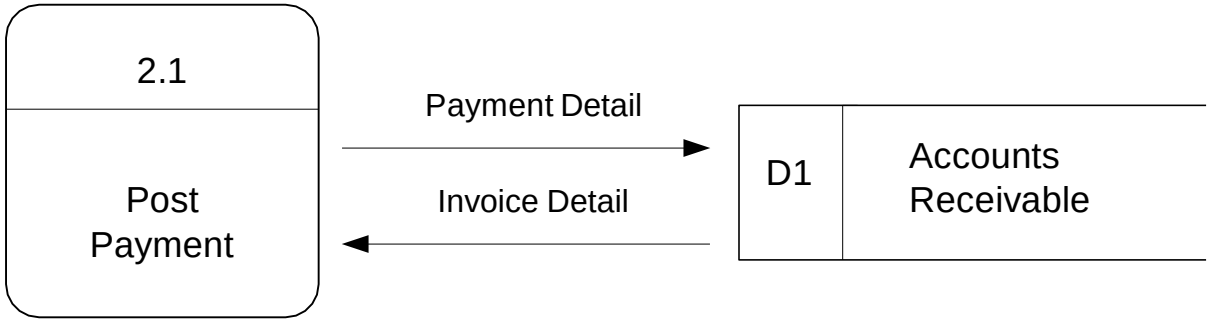
- External Entity →Noun
- Data Flow →Names of data
- Process→verbphrase
- Data Store →Noun

Process



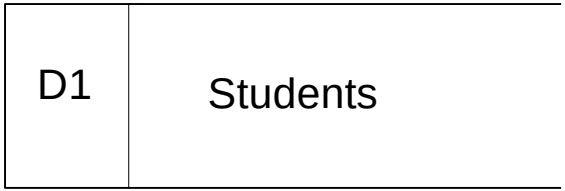
- The work or actions performed on data so that they are transformed, stored, or distributed.
- Process labels should be verb phrases!

Data Flow



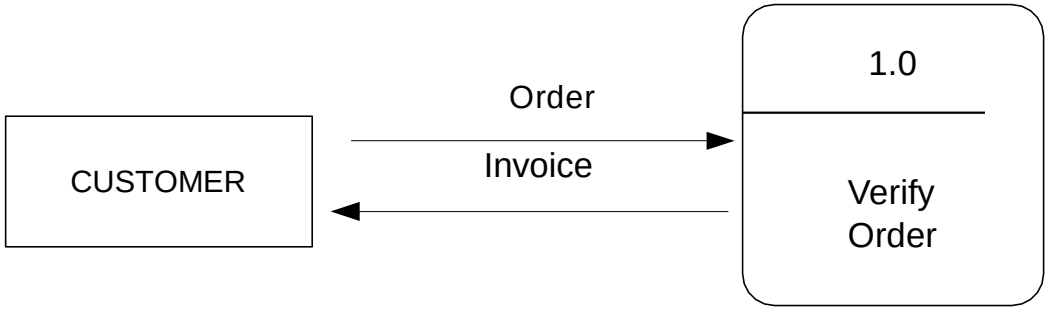
- A path for data to move from one part of the system to another.
- Data in motion!
- Arrows depict the movement of data.

Data Store



- Used in a DFD to represent data that the system stores
- Data at rest!
- Labels should be noun phrases

External Entity Source/Sink



- The origin or destination of data! – This represents things outside of the system.
- Source – Entity that supplies data to the system.
- Sink – Entity that receives data from the system.

Advantages of DFDs

- Simple graphical techniques which are easy to understand
- Helps define the boundaries of the system
- Useful for communicating current system knowledge to users
- Explains the logic behind the data flow within the system
- Used as the part of system documentation file

Decomposition Of DFD

Levels	Description	Explanation
Level 0	Context diagram	Contains only one process
Level 1	Overview diagram	Utilizes all four elements
Level 2	Detailed diagram	A breakdown of a level 2 process

Note: There is no rule as to how many levels of DFD that can be used.

Creating Data Flow Diagrams

Creating DFDs is a highly iterative process of gradual refinement.

General steps:

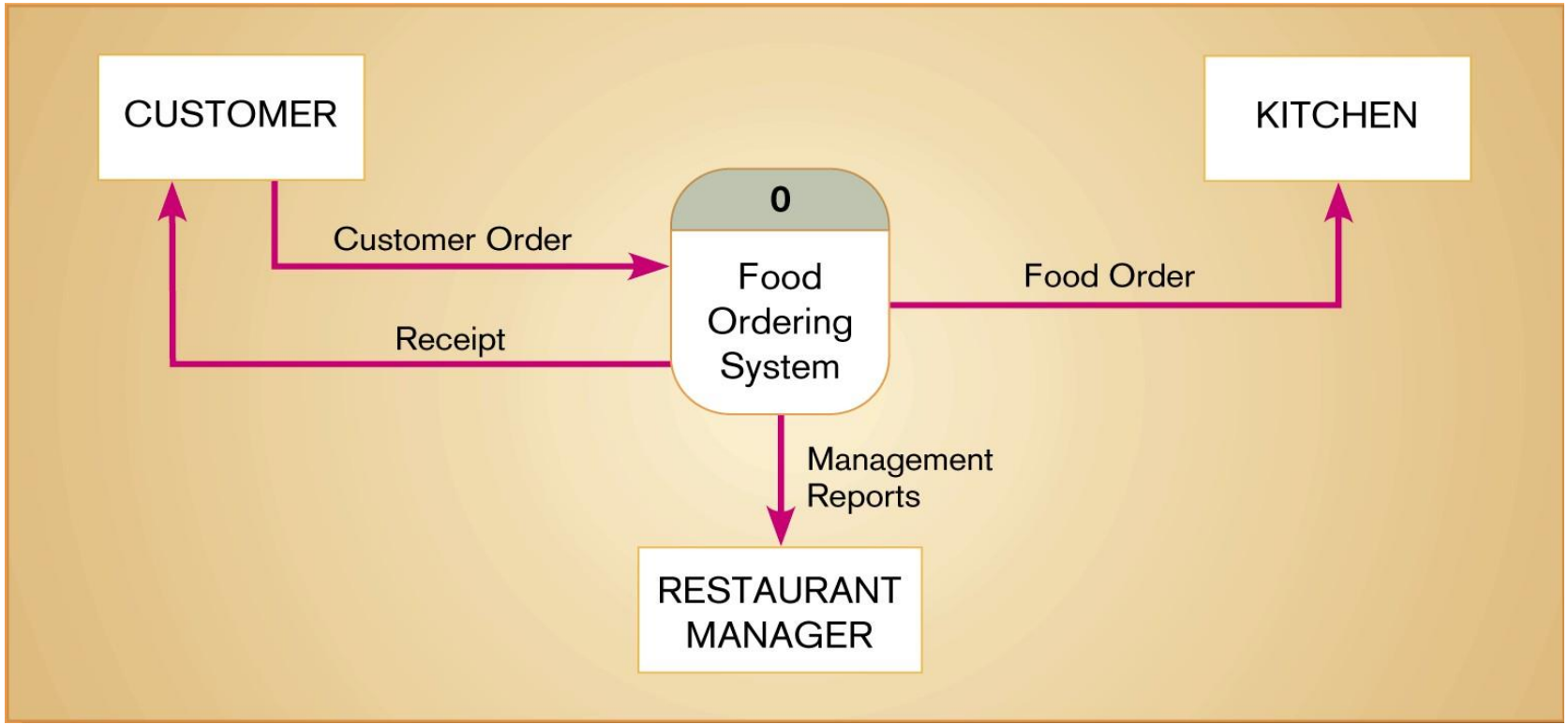
1. Create a preliminary Context Diagram
2. Identify Use Cases, i.e. the ways in which users most commonly use the system
3. Create DFD fragments for each use case
4. Create a Level 0 diagram from fragments
5. Decompose to Level 1,2,…
6. Go to step 1 and revise as necessary
7. Validate DFDs with users.

Types of Diagrams

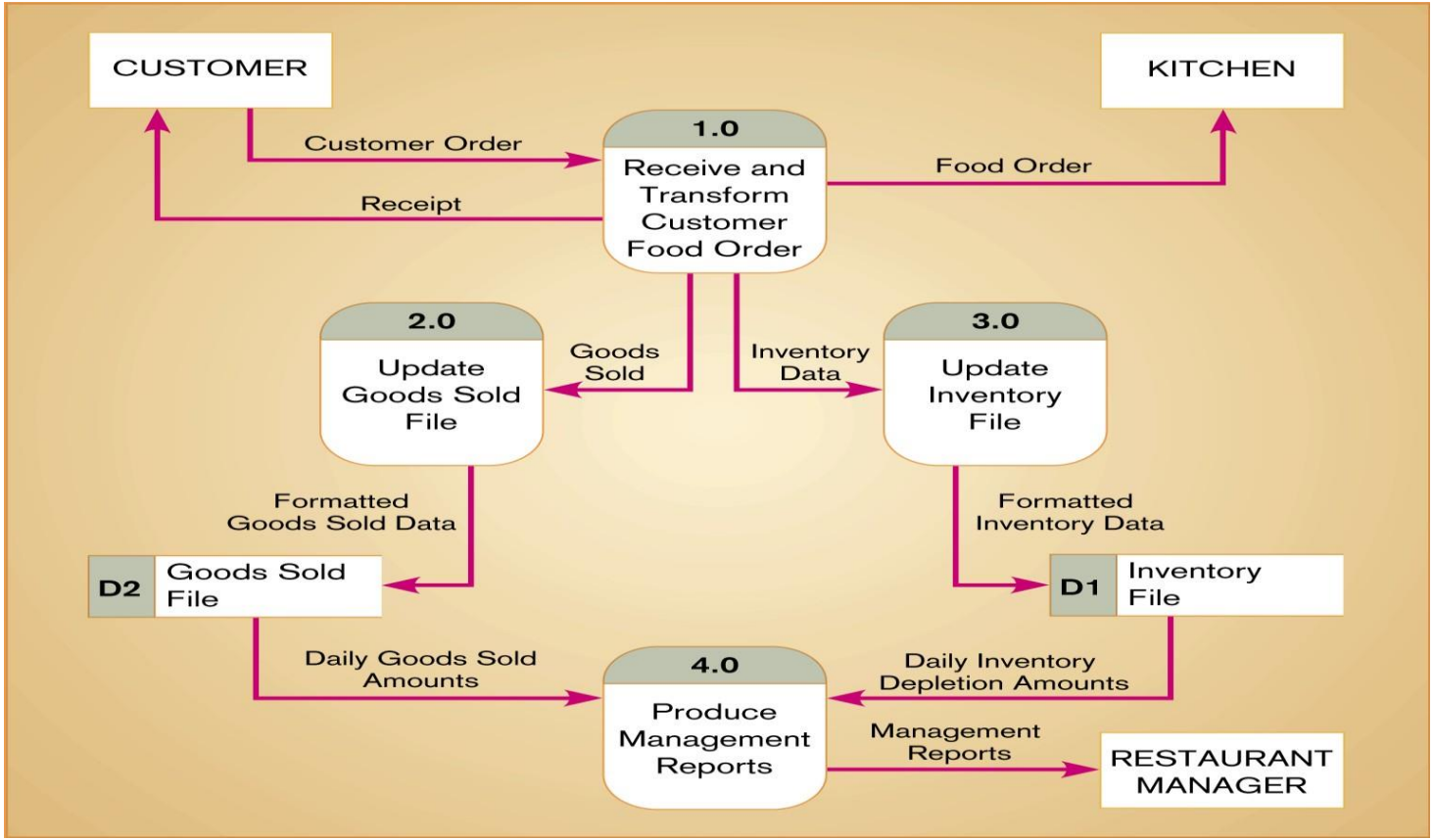
- Context Diagram
  - A data flow diagram (DFD) of the scope of an organizational system that shows the system boundaries, external entities that interact with the system and the major information flows between the entities and the system

- Level-O Diagram
  - A data flow diagram (DFD) that represents a system's major processes, data flows and data stores at a high level of detail

Context diagram of Burger's Food ordering syst



Level-0 DFD of food ordering system



A Context Diagram (Level 0)

- The major information flows between the entities and the system.
- A Context Diagram addresses only one process.

Rules for Level 0 Diagram

- 1 process represents the entire system
- Data arrows show input and output
- Data Stores NOT shown. They are within the system.

Rules for Level 1 Diagram

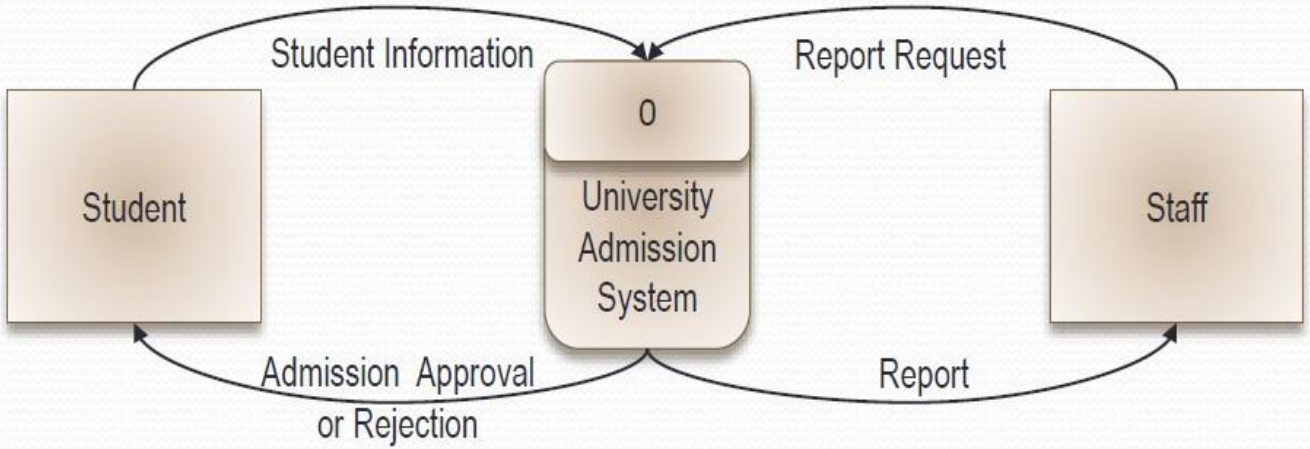
- Level 1 DFD, must balance with the context diagram it describes.
- Input going into a process are different from outputs leaving the process.
- Data stores are first shown at this level.

Rules for Level 2 Diagram

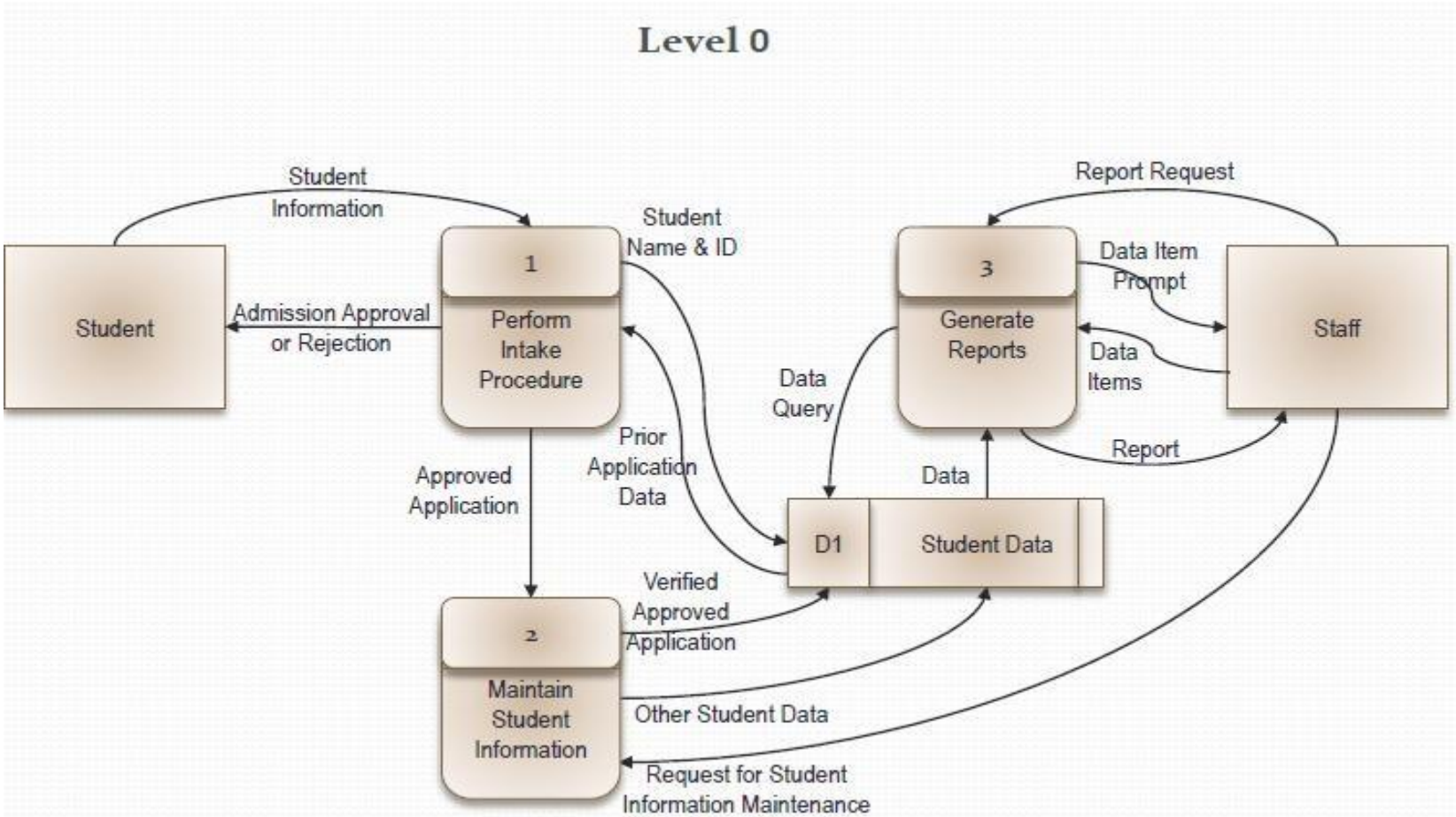
- Level 2 DFD must balance with the Level 1 it describes.
- Input going into a process are different from outputs leaving the process.
- Continue to show data stores.

DFD for University Admission System

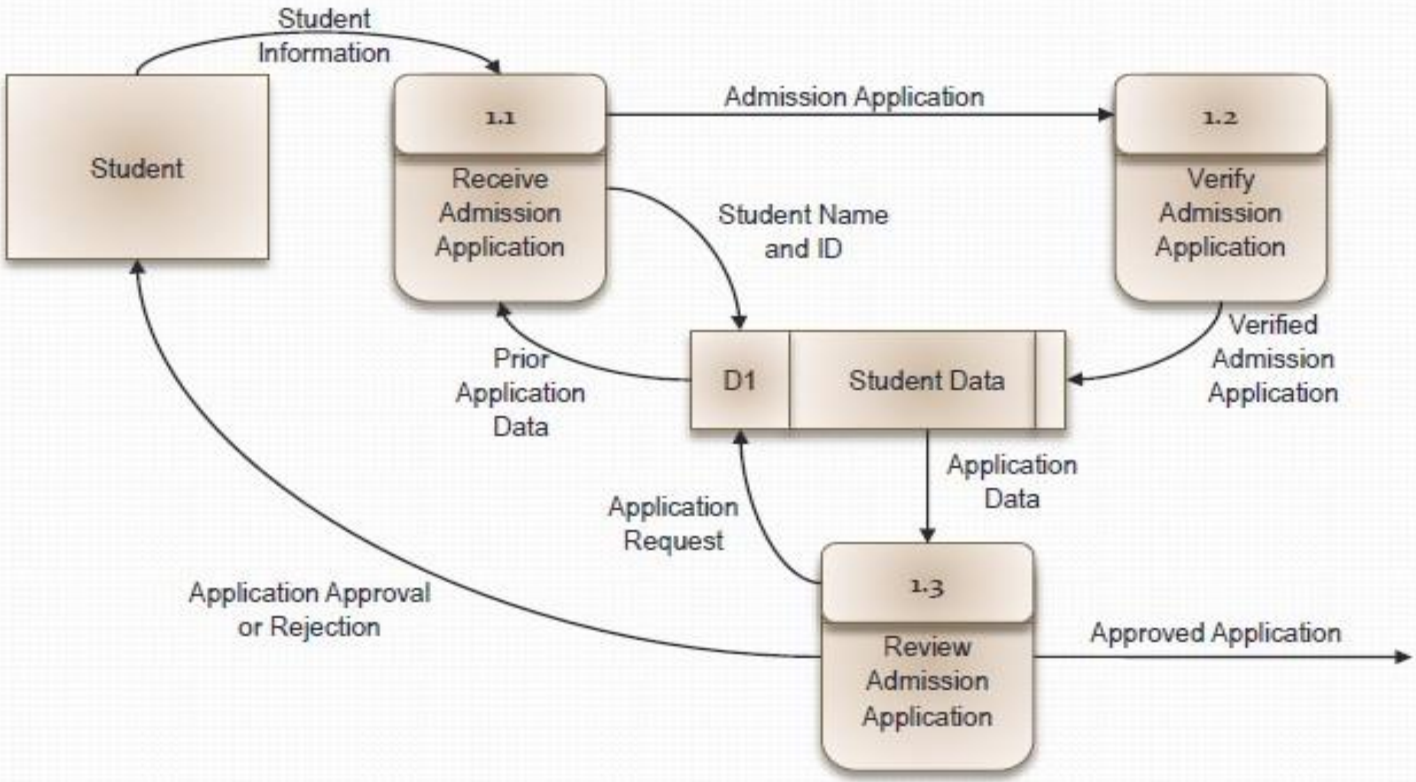
Context Diagram



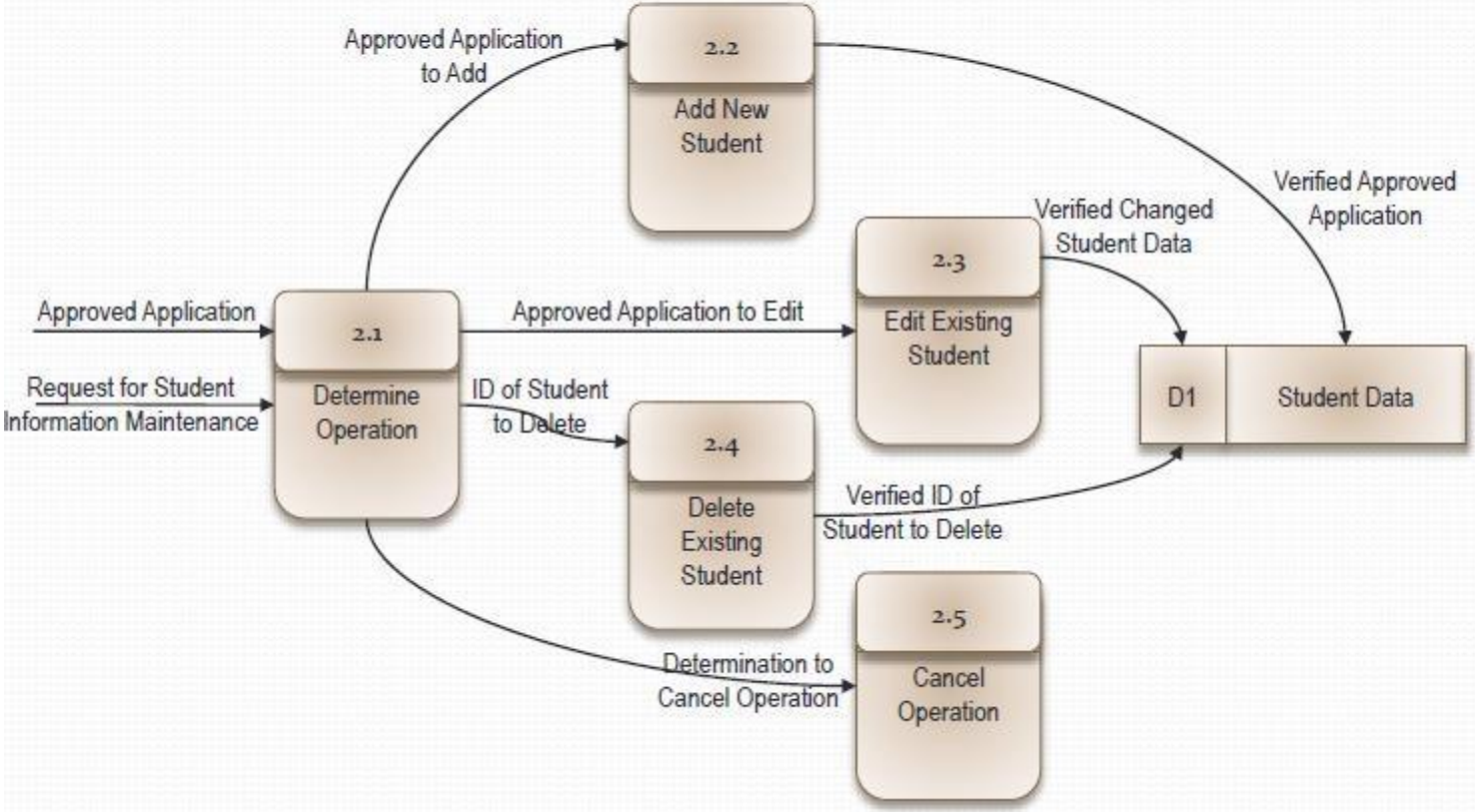
Level 0


Student
Information
Student
Name & ID
1
Perform
Intake
Procedure
Admission Approval
or Rejection
Student
Report Request
3
Generate
Reports
Data Item
Prompt
Staff
Data
Items
Report
Data
Query
Prior
Application
Data
Approved
Application
D1
Student Data
Data
Verified
Approved
Application
2
Maintain
Student
Information
Other Student Data
Request for Student
Information Maintenance

Level 1 Process 1, Perform Intake Procedure



Level 1 Process 2, Maintain Student Information



Object oriented programming style:

Reusability (Write Once and Run any where)

- Java is a programming language originally developed by Sun Microsystems that is well known for its heavily object-oriented design and nearly complete cross-platform abilities.
- Software reusability stems from the idea that a programming problem should only be solved once and, from then on, the code for the solution simply copied into a project that needs it.
- Reusability in Java means that once you write some code, you can use it over and over again - if you write it so that it can be reused.

Object oriented programming style:

Extensibility

- In software engineering, extensibility is a system design principle where the implementation takes into consideration future growth.
- It is a systemic measure of the ability to extend a system and the level of effort required to implement the extension.
- Extensions can be through the addition of new functionality or through modification of existing functionality.
- The central theme is to provide for change – typically enhancements – while minimizing impact to existing system functions

Object oriented programming style:

- **Robustness**

  consider two of the main reasons for program failure: memory management mistakes and mishandled exceptional conditions.
- **Memory management :** de allocation is completely automatic, because Java provides garbage collection for unused objects.
- **Mishandled exceptional:** Java program, all run-time errors can—and should— be managed by your program.

Thanks