

Unit 7: Transition Analysis to Design using Interaction Diagrams

Most Essential Learning Competencies: *At the end of the course, you must be able to:*

1. Discuss and define object-oriented paradigm, object-oriented analysis and object-oriented design
2. Explain the concepts of object-oriented programming
3. Enumerate different analysis techniques



Reading Activity

Understanding the Elements & Principles of Design

Design is about making decisions with purpose and organizing or placing elements in the most appropriate manner in order to communicate a message. These actions require a thorough knowledge of the elements and principles of design and a well-planned procedure.

Try to fix a car or put together a barbecue without knowing what a car or barbecue is used for. Take a trip to the downtown area of a large, unfamiliar city in Poland without a map to find a shop that sells fresh blintzes.

You can quickly see how important knowledge and research are if you are trying to find a solution to a problem and develop a plan to get it done.

The elements of design are what are used to create a composition (lines, shapes, value, etc.), while the principles of design are how the elements are used. Because the principles of design are based on psychological studies of people's perceptions of visual elements, they provide us with a basic framework for applying the elements of design in order to achieve a desired mood or effect. These psychological studies – i.e. understandings of how the eye and mind work together to perceive and organize visual information – have been linked to the Gestalt Theory, which infers two fundamental concepts:

- People inherently look for order or a relationship between various elements
- People perceive individual parts of an image as separate components, which are readily grouped as part of a larger whole

Designers use the way people naturally organize images in their heads in order to emphasize particular aspects of a composition. A successful design is one that uses the principles of design to arrange the elements of design in a way that creates a desired effect based on the Gestalt Theory. The core principles of design include unity/harmony, variety, focal point/visual hierarchy, and balance. Unity can be achieved via placement using repetition of line or shape, rhythm, value, or continuity while variety can be created with contrasting lines and shapes based on visual weight, size, and type. Repetition, rhythm, continuity, variety, scale, proximity, and contrast, therefore, can be considered principles of design as well.

There is a significant amount of overlapping of principles and applications of techniques that can be used to create a desired effect, and the principles of design really can be found in the organization and design of almost anything.

A Brief History

The object-oriented paradigm took its shape from the initial concept of a new programming approach, while the interest in design and analysis methods came much later.

- The first object-oriented language was Simula (Simulation of real systems) that was developed in 1960 by researchers at the Norwegian Computing Center.
- In 1970, Alan Kay and his research group at Xerox PARK created a personal computer named Dynabook and the first pure object-oriented programming language (OOPL) - Smalltalk, for programming the Dynabook.
- In the 1980s, Grady Booch published a paper titled Object Oriented Design that mainly presented a design for the programming language, Ada. In the ensuing editions, he extended his ideas to a complete object-oriented design method.
- In the 1990s, Coad incorporated behavioral ideas to object-oriented methods.

The other significant innovations were Object Modelling Techniques (OMT) by James Rumbaugh and Object-Oriented Software Engineering (OOSE) by Ivar Jacobson.

Object-Oriented Analysis

Object-Oriented Analysis (OOA) is the procedure of identifying software engineering requirements and developing software specifications in terms of a software system's object model, which comprises of interacting objects.

The main difference between object-oriented analysis and other forms of analysis is that in object-oriented approach, requirements are organized around objects, which integrate both data and functions. They are modelled after real-world objects that the system interacts with. In traditional analysis methodologies, the two aspects - functions and data - are considered separately.

Grady Booch has defined OOA as, "Object-oriented analysis is a method of analysis that examines requirements from the perspective of the classes and objects found in the vocabulary of the problem domain".

The primary tasks in object-oriented analysis (OOA) are –

- Identifying objects
- Organizing the objects by creating object model diagram
- Defining the internals of the objects, or object attributes
- Defining the behavior of the objects, i.e., object actions
- Describing how the objects interact

The common models used in OOA are use cases and object models.

Object-Oriented Design

Object-Oriented Design (OOD) involves implementation of the conceptual model produced during object-oriented analysis. In OOD, concepts in the analysis model, which are technology-independent, are mapped onto implementing classes, constraints are identified and interfaces are designed, resulting

in a model for the solution domain, i.e., a detailed description of how the system is to be built on concrete technologies.

The implementation details generally include –

- Restructuring the class data (if necessary),
- Implementation of methods, i.e., internal data structures and algorithms,
- Implementation of control, and
- Implementation of associations.

Grady Booch has defined object-oriented design as “a method of design encompassing the process of object-oriented decomposition and a notation for depicting both logical and physical as well as static and dynamic models of the system under design”.

Object-Oriented Programming

Object-oriented programming (OOP) is a programming paradigm based upon objects (having both data and methods) that aims to incorporate the advantages of modularity and reusability. Objects, which are usually instances of classes, are used to interact with one another to design applications and computer programs.

The important features of object-oriented programming are –

- Bottom-up approach in program design
- Programs organized around objects, grouped in classes
- Focus on data with methods to operate upon object's data
- Interaction between objects through functions
- Reusability of design through creation of new classes by adding features to existing classes

Some examples of object-oriented programming languages are C++, Java, Smalltalk, Delphi, C#, Perl, Python, Ruby, and PHP.

Grady Booch has defined object-oriented programming as “a method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships”.

The object model visualizes the elements in a software application in terms of objects. In this chapter, we will look into the basic concepts and terminologies of object-oriented systems.

Objects and Classes

The concepts of objects and classes are intrinsically linked with each other and form the foundation of object-oriented paradigm.

Object

An object is a real-world element in an object-oriented environment that may have a physical or a conceptual existence. Each object has –

- Identity that distinguishes it from other objects in the system.
- State that determines the characteristic properties of an object as well as the values of the properties that the object holds.

- Behavior that represents externally visible activities performed by an object in terms of changes in its state.

Objects can be modelled according to the needs of the application. An object may have a physical existence, like a customer, a car, etc.; or an intangible conceptual existence, like a project, a process, etc.

Class

A class represents a collection of objects having same characteristic properties that exhibit common behavior. It gives the blueprint or description of the objects that can be created from it. Creation of an object as a member of a class is called instantiation. Thus, object is an instance of a class.

The constituents of a class are –

- A set of attributes for the objects that are to be instantiated from the class. Generally, different objects of a class have some difference in the values of the attributes. Attributes are often referred as class data.
- A set of operations that portray the behavior of the objects of the class. Operations are also referred as functions or methods.

Example

Let us consider a simple class, Circle, that represents the geometrical figure circle in a two-dimensional space. The attributes of this class can be identified as follows –

- x-coord, to denote x-coordinate of the center
- y-coord, to denote y-coordinate of the center
- a, to denote the radius of the circle

Some of its operations can be defined as follows –

- findArea(), method to calculate area
- findCircumference(), method to calculate circumference
- scale(), method to increase or decrease the radius

During instantiation, values are assigned for at least some of the attributes. If we create an object my_circle, we can assign values like x-coord : 2, y-coord : 3, and a : 4 to depict its state. Now, if the operation scale() is performed on my_circle with a scaling factor of 2, the value of the variable a will become 8. This operation brings a change in the state of my_circle, i.e., the object has exhibited certain behavior.

Encapsulation and Data Hiding

Encapsulation is the process of binding both attributes and methods together within a class. Through encapsulation, the internal details of a class can be hidden from outside. It permits the elements of the class to be accessed from outside only through the interface provided by the class.

Typically, a class is designed such that its data (attributes) can be accessed only by its class methods and insulated from direct outside access. This process of insulating an object's data is called data hiding or information hiding.

Example

In the class Circle, data hiding can be incorporated by making attributes invisible from outside the class and adding two more methods to the class for accessing class data, namely –

- `setValues()`, method to assign values to x-coord, y-coord, and a
- `getValues()`, method to retrieve values of x-coord, y-coord, and a

Here the private data of the object `my_circle` cannot be accessed directly by any method that is not encapsulated within the class `Circle`. It should instead be accessed through the methods `setValues()` and `getValues()`.

Message Passing

Any application requires a number of objects interacting in a harmonious manner. Objects in a system may communicate with each other using message passing. Suppose a system has two objects: `obj1` and `obj2`. The object `obj1` sends a message to object `obj2`, if `obj1` wants `obj2` to execute one of its methods. The features of message passing are –

- Message passing between two objects is generally unidirectional.
- Message passing enables all interactions between objects.
- Message passing essentially involves invoking class methods.
- Objects in different processes can be involved in message passing.

Inheritance

Inheritance is the mechanism that permits new classes to be created out of existing classes by extending and refining its capabilities. The existing classes are called the base classes/parent classes/super-classes, and the new classes are called the derived classes/child classes/subclasses. The subclass can inherit or derive the attributes and methods of the super-class(es) provided that the super-class allows so. Besides, the subclass may add its own attributes and methods and may modify any of the super-class methods. Inheritance defines an “is – a” relationship.

Example

From a class `Mammal`, a number of classes can be derived such as `Human`, `Cat`, `Dog`, `Cow`, etc. Humans, cats, dogs, and cows all have the distinct characteristics of mammals. In addition, each has its own particular characteristics. It can be said that a cow “is – a” mammal.

Types of Inheritance

- **Single Inheritance** – A subclass derives from a single super-class.
- **Multiple Inheritance** – A subclass derives from more than one super-classes.
- **Multilevel Inheritance** – A subclass derives from a super-class which in turn is derived from another class and so on.
- **Hierarchical Inheritance** – A class has a number of subclasses each of which may have subsequent subclasses, continuing for a number of levels, so as to form a tree structure.
- **Hybrid Inheritance** – A combination of multiple and multilevel inheritance so as to form a lattice structure.

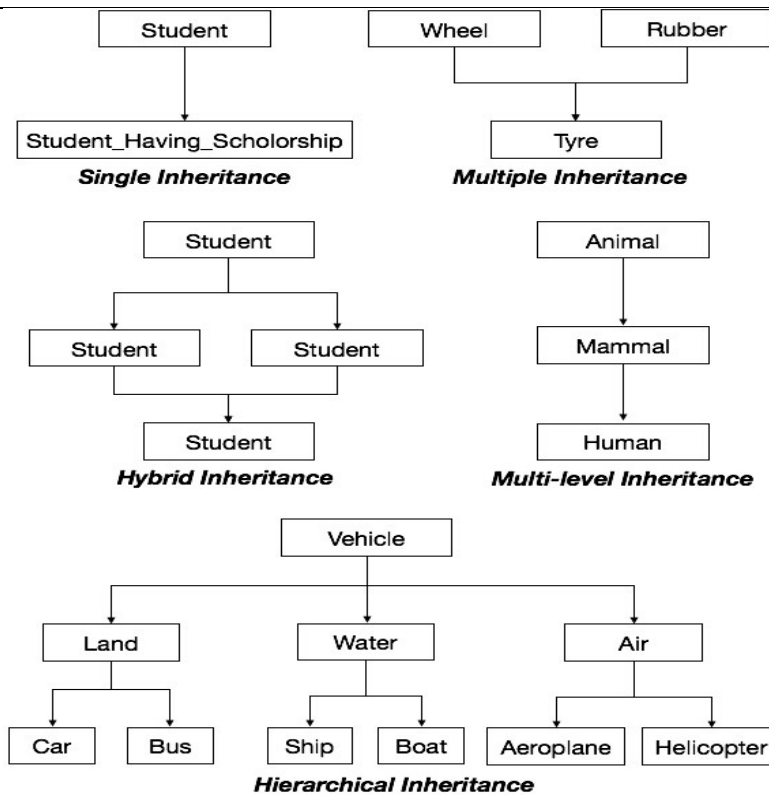


Figure 1: The figure depicts the examples of different types of inheritance.

Polymorphism

Polymorphism is originally a Greek word that means the ability to take multiple forms. In object-oriented paradigm, polymorphism implies using operations in different ways, depending upon the instance they are operating upon. Polymorphism allows objects with different internal structures to have a common external interface. Polymorphism is particularly effective while implementing inheritance.

Example

Let us consider two classes, Circle and Square, each with a method findArea(). Though the name and purpose of the methods in the classes are same, the internal implementation, i.e., the procedure of calculating area is different for each class. When an object of class Circle invokes its findArea() method, the operation finds the area of the circle without any conflict with the findArea() method of the Square class.

Generalization and Specialization

Generalization and specialization represent a hierarchy of relationships between classes, where subclasses inherit from super-classes.

In the generalization process, the common characteristics of classes are combined to form a class in a higher level of hierarchy, i.e., subclasses are combined to form a generalized super-class. It represents

an “is – a – kind – of” relationship. For example, “car is a kind of land vehicle”, or “ship is a kind of water vehicle”.

Specialization is the reverse process of generalization. Here, the distinguishing features of groups of objects are used to form specialized classes from existing classes. It can be said that the subclasses are the specialized versions of the super-class.

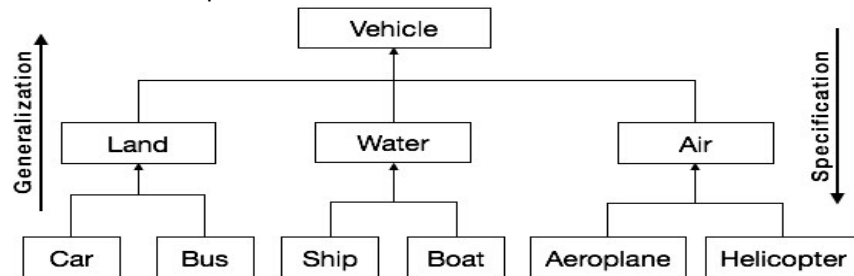


Figure 2: The following figure shows an example of generalization and specialization.

Links and Association

A link represents a connection through which an object collaborates with other objects. Rumbaugh has defined it as “a physical or conceptual connection between objects”. Through a link, one object may invoke the methods or navigate through another object. A link depicts the relationship between two or more objects.

Association is a group of links having common structure and common behavior. Association depicts the relationship between objects of one or more classes. A link can be defined as an instance of an association.

Degree of an Association

Degree of an association denotes the number of classes involved in a connection. Degree may be unary, binary, or ternary.

- A unary relationship connects objects of the same class.
- A binary relationship connects objects of two classes.
- A ternary relationship connects objects of three or more classes.

Cardinality Ratios of Associations

Cardinality of a binary association denotes the number of instances participating in an association. There are three types of cardinality ratios, namely –

- One-to-One – A single object of class A is associated with a single object of class B.
- One-to-Many – A single object of class A is associated with many objects of class B.
- Many-to-Many – An object of class A may be associated with many objects of class B and conversely an object of class B may be associated with many objects of class A.

Aggregation or Composition

Aggregation or composition is a relationship among classes by which a class can be made up of any combination of objects of other classes. It allows objects to be placed directly within the body of other classes. Aggregation is referred as a “part-of” or “has-a” relationship, with the ability to navigate from the whole to its parts. An aggregate object is an object that is composed of one or more other objects.

Example

In the relationship, “a car has-a motor”, car is the whole object or the aggregate, and the motor is a “part-of” the car. Aggregation may denote –

- Physical containment – Example, a computer is composed of monitor, CPU, mouse, keyboard, and so on.
- Conceptual containment – Example, shareholder has-a share.

Benefits of Object Model

Now that we have gone through the core concepts pertaining to object orientation, it would be worthwhile to note the advantages that this model has to offer.

The benefits of using the object model are –

- It helps in faster development of software.
- It is easy to maintain. Suppose a module develops an error, then a programmer can fix that particular module, while the other parts of the software are still up and running.
- It supports relatively hassle-free upgrades.
- It enables reuse of objects, designs, and functions.
- It reduces development risks, particularly in integration of complex systems.

We know that the Object-Oriented Modelling (OOM) technique visualizes things in an application by using models organized around objects. Any software development approach goes through the following stages –

- Analysis,
- Design, and
- Implementation.

In object-oriented software engineering, the software developer identifies and organizes the application in terms of object-oriented concepts, prior to their final representation in any specific programming language or software tools.

Phases in Object-Oriented Software Development

The major phases of software development using object-oriented methodology are object-oriented analysis, object-oriented design, and object-oriented implementation.

Object–Oriented Analysis

In this stage, the problem is formulated, user requirements are identified, and then a model is built based upon real–world objects. The analysis produces models on how the desired system should function and how it must be developed. The models do not include any implementation details so that it can be understood and examined by any non–technical application expert.

Object–Oriented Design

Object-oriented design includes two main stages, namely, system design and object design.

System Design

In this stage, the complete architecture of the desired system is designed. The system is conceived as a set of interacting subsystems that in turn is composed of a hierarchy of interacting objects, grouped into classes. System design is done according to both the system analysis model and the proposed system architecture. Here, the emphasis is on the objects comprising the system rather than the processes in the system.

Object Design

In this phase, a design model is developed based on both the models developed in the system analysis phase and the architecture designed in the system design phase. All the classes required are identified. The designer decides whether –

- new classes are to be created from scratch,
- any existing classes can be used in their original form, or
- new classes should be inherited from the existing classes.

The associations between the identified classes are established and the hierarchies of classes are identified. Besides, the developer designs the internal details of the classes and their associations, i.e., the data structure for each attribute and the algorithms for the operations.

Object–Oriented Implementation and Testing

In this stage, the design model developed in the object design is translated into code in an appropriate programming language or software tool. The databases are created and the specific hardware requirements are ascertained. Once the code is in shape, it is tested using specialized techniques to identify and remove the errors in the code.

Principles of Object-Oriented Systems

The conceptual framework of object–oriented systems is based upon the object model. There are two categories of elements in an object-oriented system –

Major Elements – By major, it is meant that if a model does not have any one of these elements, it ceases to be object oriented. The four major elements are –

- Abstraction
- Encapsulation
- Modularity
- Hierarchy

Minor Elements – By minor, it is meant that these elements are useful, but not indispensable part of the object model. The three minor elements are –

- Typing
- Concurrency
- Persistence

Abstraction

Abstraction means to focus on the essential features of an element or object in OOP, ignoring its extraneous or accidental properties. The essential features are relative to the context in which the object is being used.

Grady Booch has defined abstraction as follows –

“An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer.”

Example – When a class Student is designed, the attributes enrolment_number, name, course, and address are included while characteristics like pulse_rate and size_of_shoe are eliminated, since they are irrelevant in the perspective of the educational institution.

Encapsulation

Encapsulation is the process of binding both attributes and methods together within a class. Through encapsulation, the internal details of a class can be hidden from outside. The class has methods that provide user interfaces by which the services provided by the class may be used.

Modularity

Modularity is the process of decomposing a problem (program) into a set of modules so as to reduce the overall complexity of the problem. Booch has defined modularity as –

“Modularity is the property of a system that has been decomposed into a set of cohesive and loosely coupled modules.”

Modularity is intrinsically linked with encapsulation. Modularity can be visualized as a way of mapping encapsulated abstractions into real, physical modules having high cohesion within the modules and their inter-module interaction or coupling is low.

Hierarchy

In Grady Booch's words, "Hierarchy is the ranking or ordering of abstraction". Through hierarchy, a system can be made up of interrelated subsystems, which can have their own subsystems and so on until the smallest level components are reached. It uses the principle of "divide and conquer". Hierarchy allows code reusability.

The two types of hierarchies in OOA are –

- "IS-A" hierarchy – It defines the hierarchical relationship in inheritance, whereby from a super-class, a number of subclasses may be derived which may again have subclasses and so on. For example, if we derive a class Rose from a class Flower, we can say that a rose "is-a" flower.
- "PART-OF" hierarchy – It defines the hierarchical relationship in aggregation by which a class may be composed of other classes. For example, a flower is composed of sepals, petals, stamens, and carpel. It can be said that a petal is a "part-of" flower.

Typing

According to the theories of abstract data type, a type is a characterization of a set of elements. In OOP, a class is visualized as a type having properties distinct from any other types. Typing is the enforcement of the notion that an object is an instance of a single class or type. It also enforces that objects of different types may not be generally interchanged; and can be interchanged only in a very restricted manner if absolutely required to do so.

The two types of typing are –

- Strong Typing – Here, the operation on an object is checked at the time of compilation, as in the programming language Eiffel.
- Weak Typing – Here, messages may be sent to any class. The operation is checked only at the time of execution, as in the programming language Smalltalk.

Concurrency

Concurrency in operating systems allows performing multiple tasks or processes simultaneously. When a single process exists in a system, it is said that there is a single thread of control. However, most systems have multiple threads, some active, some waiting for CPU, some suspended, and some terminated. Systems with multiple CPUs inherently permit concurrent threads of control; but systems running on a single CPU use appropriate algorithms to give equitable CPU time to the threads so as to enable concurrency.

In an object-oriented environment, there are active and inactive objects. The active objects have independent threads of control that can execute concurrently with threads of other objects. The active objects synchronize with one another as well as with purely sequential objects.

Persistence

An object occupies a memory space and exists for a particular period of time. In traditional programming, the lifespan of an object was typically the lifespan of the execution of the program that created it. In files or databases, the object lifespan is longer than the duration of the process creating the object. This property by which an object continues to exist even after its creator ceases to exist is known as persistence.

In the system analysis or object-oriented analysis phase of software development, the system requirements are determined, the classes are identified and the relationships among classes are identified.

The three analysis techniques that are used in conjunction with each other for object-oriented analysis are object modelling, dynamic modelling, and functional modelling.

Object Modelling

Object modelling develops the static structure of the software system in terms of objects. It identifies the objects, the classes into which the objects can be grouped into and the relationships between the objects. It also identifies the main attributes and operations that characterize each class.

The process of object modelling can be visualized in the following steps –

- Identify objects and group into classes
- Identify the relationships among classes
- Create user object model diagram
- Define user object attributes
- Define the operations that should be performed on the classes
- Review glossary

Dynamic Modelling

After the static behavior of the system is analyzed, its behavior with respect to time and external changes needs to be examined. This is the purpose of dynamic modelling.

Dynamic Modelling can be defined as “a way of describing how an individual object responds to events, either internal events triggered by other objects, or external events triggered by the outside world”.

The process of dynamic modelling can be visualized in the following steps –

- Identify states of each object
- Identify events and analyze the applicability of actions
- Construct dynamic model diagram, comprising of state transition diagrams
- Express each state in terms of object attributes
- Validate the state–transition diagrams drawn

Functional Modelling

Functional Modelling is the final component of object-oriented analysis. The functional model shows the processes that are performed within an object and how the data changes as it moves between methods. It specifies the meaning of the operations of object modelling and the actions of dynamic modelling. The functional model corresponds to the data flow diagram of traditional structured analysis.

The process of functional modelling can be visualized in the following steps –

- Identify all the inputs and outputs
- Construct data flow diagrams showing functional dependencies
- State the purpose of each function
- Identify constraints
- Specify optimization criteria

Structured Analysis vs. Object Oriented Analysis

The Structured Analysis/Structured Design (SASD) approach is the traditional approach of software development based upon the waterfall model. The phases of development of a system using SASD are :

- Feasibility Study
- Requirement Analysis and Specification
- System Design
- Implementation
- Post-implementation Review

Now, we will look at the relative advantages and disadvantages of structured analysis approach and object-oriented analysis approach.

Advantages/Disadvantages of Object-Oriented Analysis

Advantages	Disadvantages
Focuses on data rather than the procedures as in Structured Analysis.	Functionality is restricted within objects. This may pose a problem for systems which are intrinsically procedural or computational in nature.
The principles of encapsulation and data hiding help the developer to develop systems that cannot be tampered by other parts of the system.	It cannot identify which objects would generate an optimal system design.
The principles of encapsulation and data hiding help the developer to develop systems that cannot be tampered by other parts of the system.	The object-oriented models do not easily show the communications between the objects in the system.
It allows effective management of software	All the interfaces between the objects cannot be

complexity by the virtue of modularity.	represented in a single diagram.
It can be upgraded from small to large systems at a greater ease than in systems following structured analysis.	

Advantages/Disadvantages of Structured Analysis

Advantages	Disadvantages
As it follows a top-down approach in contrast to bottom-up approach of object-oriented analysis, it can be more easily comprehended than OOA.	In traditional structured analysis models, one phase should be completed before the next phase. This poses a problem in design, particularly if errors crop up or requirements change.
It is based upon functionality. The overall purpose is identified and then functional decomposition is done for developing the software. The emphasis not only gives a better understanding of the system but also generates more complete systems.	The initial cost of constructing the system is high, since the whole system needs to be designed at once leaving very little option to add functionality later.
The specifications in it are written in simple English language, and hence can be more easily analyzed by non-technical personnel.	It does not support reusability of code. So, the time and cost of development is inherently high.

The dynamic model represents the time-dependent aspects of a system. It is concerned with the temporal changes in the states of the objects in a system. The main concepts are –

- State, which is the situation at a particular condition during the lifetime of an object.
- Transition, a change in the state
- Event, an occurrence that triggers transitions
- Action, an uninterrupted and atomic computation that occurs due to some event, and
- Concurrency of transitions.

A state machine models the behavior of an object as it passes through a number of states in its lifetime due to some events as well as the actions occurring due to the events. A state machine is graphically represented through a state transition diagram.

States and State Transitions

The state is an abstraction given by the values of the attributes that the object has at a particular time period. It is a situation occurring for a finite time period in the lifetime of an object, in which it fulfils certain conditions, performs certain activities, or waits for certain events to occur. In state transition diagrams, a state is represented by rounded rectangles.

Parts of a state

- Name – A string differentiates one state from another. A state may not have any name.
- Entry/Exit Actions – It denotes the activities performed on entering and on exiting the state.
- Internal Transitions – The changes within a state that do not cause a change in the state.
- Sub-states – States within states.

Initial and Final States

The default starting state of an object is called its initial state. The final state indicates the completion of execution of the state machine. The initial and the final states are pseudo-states, and may not have the parts of a regular state except name. In state transition diagrams, the initial state is represented by a filled black circle. The final state is represented by a filled black circle encircled within another unfilled black circle.

Transition

A transition denotes a change in the state of an object. If an object is in a certain state when an event occurs, the object may perform certain activities subject to specified conditions and change the state. In this case, a state-transition is said to have occurred. The transition gives the relationship between the first state and the new state. A transition is graphically represented by a solid directed arc from the source state to the destination state.

The five parts of a transition are –

- Source State – The state affected by the transition.
- Event Trigger – The occurrence due to which an object in the source state undergoes a transition if the guard condition is satisfied.
- Guard Condition – A Boolean expression which if True, causes a transition on receiving the event trigger.
- Action – An un-interruptible and atomic computation that occurs on the source object due to some event.
- Target State – The destination state after completion of transition.

Example

Suppose a person is taking a taxi from place X to place Y. The states of the person may be: Waiting (waiting for taxi), Riding (he has got a taxi and is travelling in it), and Reached (he has reached the destination). The following figure depicts the state transition.

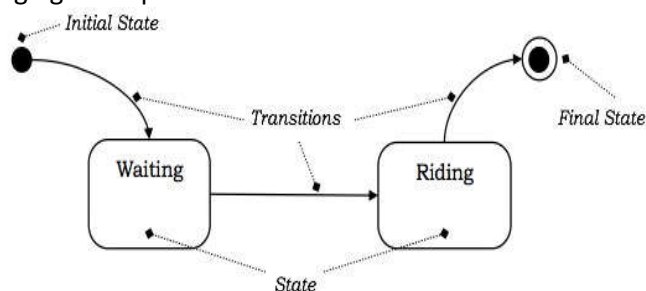


Figure 3: A representation of Transition.

Events

Events are some occurrences that can trigger state transition of an object or a group of objects. Events have a location in time and space but do not have a time period associated with it. Events are generally associated with some actions.

Examples of events are mouse click, key press, an interrupt, stack overflow, etc.

Events that trigger transitions are written alongside the arc of transition in state diagrams.

Example

Considering the example shown in the above figure, the transition from Waiting state to Riding state takes place when the person gets a taxi. Likewise, the final state is reached, when he reaches the destination. These two occurrences can be termed as events `Get_Taxi` and `Reach_Destination`.

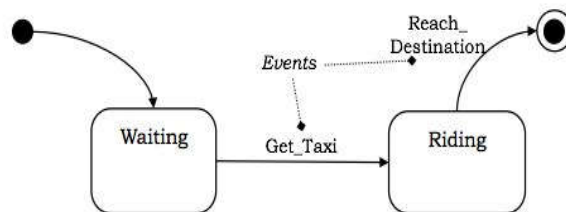


Figure 4: The above figure shows the events in a state machine.

External and Internal Events

External events are those events that pass from a user of the system to the objects within the system. For example, mouse click or key-press by the user are external events.

Internal events are those that pass from one object to another object within a system. For example, stack overflow, a divide error, etc.

Deferred Events

Deferred events are those which are not immediately handled by the object in the current state but are lined up in a queue so that they can be handled by the object in some other state at a later time.

Event Classes

Event class indicates a group of events with common structure and behavior. As with classes of objects, event classes may also be organized in a hierarchical structure. Event classes may have attributes associated with them, time being an implicit attribute. For example, we can consider the events of departure of a flight of an airline, which we can group into the following class –

`Flight_Departs (Flight_No, From_City, To_City, Route)`

Activity

Activity is an operation upon the states of an object that requires some time period. They are the ongoing executions within a system that can be interrupted. Activities are shown in activity diagrams that portray the flow from one activity to another.

Action

An action is an atomic operation that executes as a result of certain events. By atomic, it is meant that actions are un-interruptible, i.e., if an action starts executing, it runs into completion without being interrupted by any event. An action may operate upon an object on which an event has been triggered or on other objects that are visible to this object. A set of actions comprise an activity.

Entry and Exit Actions

Entry action is the action that is executed on entering a state, irrespective of the transition that led into it.

Likewise, the action that is executed while leaving a state, irrespective of the transition that led out of it, is called an exit action.

Scenario

Scenario is a description of a specified sequence of actions. It depicts the behavior of objects undergoing a specific action series. The primary scenarios depict the essential sequences and the secondary scenarios depict the alternative sequences.

Diagrams for Dynamic Modelling

There are two primary diagrams that are used for dynamic modelling –

Interaction Diagrams

Interaction diagrams describe the dynamic behavior among different objects. It comprises of a set of objects, their relationships, and the message that the objects send and receive. Thus, an interaction models the behavior of a group of interrelated objects. The two types of interaction diagrams are –

- Sequence Diagram – It represents the temporal ordering of messages in a tabular manner.
- Collaboration Diagram – It represents the structural organization of objects that send and receive messages through vertices and arcs.

State Transition Diagram

State transition diagrams or state machines describe the dynamic behavior of a single object. It illustrates the sequences of states that an object goes through in its lifetime, the transitions of the states, the events and conditions causing the transition and the responses due to the events.

Concurrency of Events

In a system, two types of concurrency may exist. They are –

- System Concurrency

Here, concurrency is modelled in the system level. The overall system is modelled as the aggregation of state machines, where each state machine executes concurrently with others.

- Concurrency within an Object

Here, an object can issue concurrent events. An object may have states that are composed of sub-states, and concurrent events may occur in each of the sub-states.

Concepts related to concurrency within an object are as follows –

Simple and Composite States

A simple state has no sub-structure. A state that has simpler states nested inside it is called a composite state. A sub-state is a state that is nested inside another state. It is generally used to reduce the complexity of a state machine. Sub-states can be nested to any number of levels.

Composite states may have either sequential sub-states or concurrent sub-states.

Sequential Sub-states

In sequential sub-states, the control of execution passes from one sub-state to another sub-state one after another in a sequential manner. There is at most one initial state and one final state in these state machines.

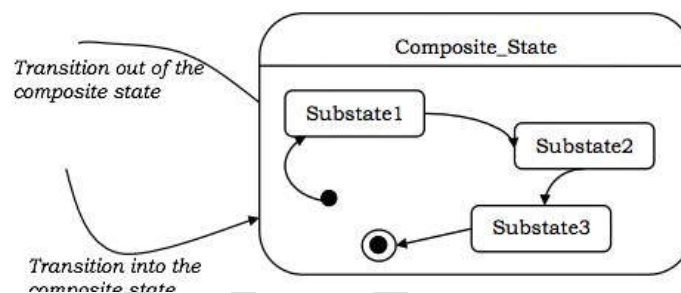


Figure 5: The figure illustrates the concept of sequential sub-states.

Concurrent Sub-states

In concurrent sub-states, the sub-states execute in parallel, or in other words, each state has concurrently executing state machines within it. Each of the state machines has its own initial and final states. If one concurrent sub-state reaches its final state before the other, control waits at its final state. When all the nested state machines reach their final states, the sub-states join back to a single flow.

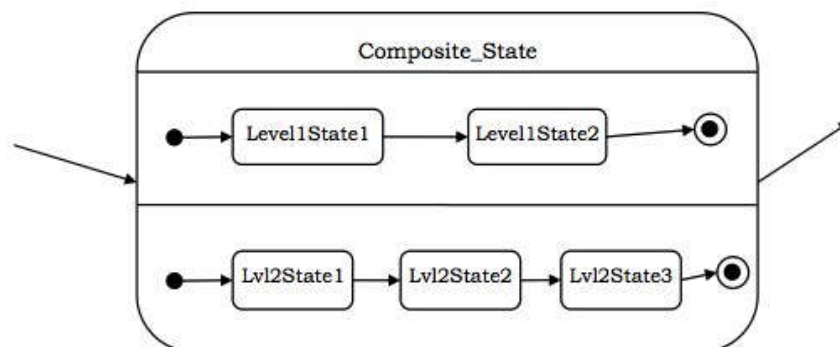


Figure 6: The figure shows the concept of concurrent sub-states.

Functional Modelling gives the process perspective of the object-oriented analysis model and an overview of what the system is supposed to do. It defines the function of the internal processes in the system with the aid of Data Flow Diagrams (DFDs). It depicts the functional derivation of the data values without indicating how they are derived when they are computed, or why they need to be computed.

Data Flow Diagrams

Functional Modelling is represented through a hierarchy of DFDs. The DFD is a graphical representation of a system that shows the inputs to the system, the processing upon the inputs, the outputs of the system as well as the internal data stores. DFDs illustrate the series of transformations or computations performed on the objects or the system, and the external controls and objects that affect the transformation.

Rumbaugh et al. have defined DFD as, “A data flow diagram is a graph which shows the flow of data values from their sources in objects through processes that transform them to their destinations on other objects.”

The four main parts of a DFD are –

- Processes,
- Data Flows,
- Actors, and
- Data Stores.

The other parts of a DFD are –

- Constraints, and
- Control Flows.

Features of a DFD

Processes - are the computational activities that transform data values.

A whole system can be visualized as a high-level process. A process may be further divided into smaller components. The lowest-level process may be a simple function.

Representation in DFD – A process is represented as an ellipse with its name written inside it and contains a fixed number of input and output data values.

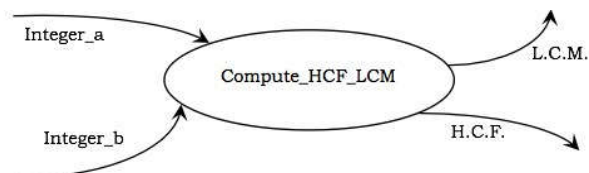


Figure 7: The figure shows a process Compute_HCF_LCM that accepts two integers as inputs and outputs their HCF (highest common factor) and LCM (least common multiple).

Data Flows

Data flow represents the flow of data between two processes. It could be between an actor and a process, or between a data store and a process. A data flow denotes the value of a data item at some point of the computation. This value is not changed by the data flow.

Representation in DFD – A data flow is represented by a directed arc or an arrow, labelled with the name of the data item that it carries.

A data flow may be forked in the following cases –

- The output value is sent to several places as shown in the following figure. Here, the output arrows are unlabeled as they denote the same value.
- The data flow contains an aggregate value, and each of the components is sent to different places as shown in the following figure. Here, each of the forked components is labelled.

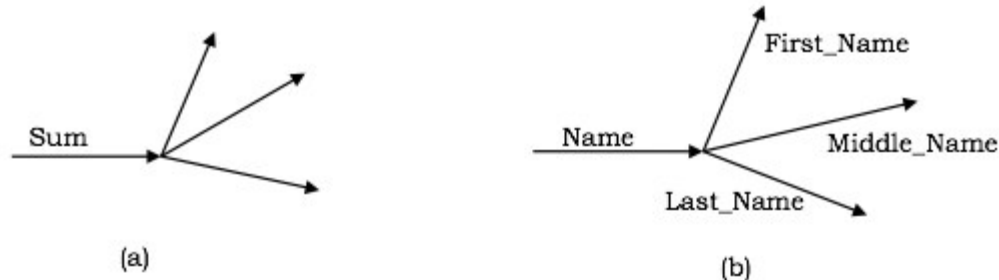


Figure 8: In the above figure, Integer_a and Integer_b represents the input data flows to the process, while L.C.M. and H.C.F. are the output data flows.

Actors

Actors are the active objects that interact with the system by either producing data and inputting them to the system, or consuming data produced by the system. In other words, actors serve as the sources and the sinks of data.

Representation in DFD – An actor is represented by a rectangle. Actors are connected to the inputs and outputs and lie on the boundary of the DFD.

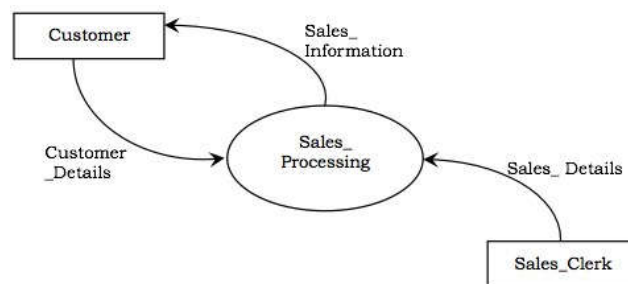


Figure 9: The figure shows the actors, namely, Customer and Sales_Clerk in a counter sales system.

Data Stores

Data stores are the passive objects that act as a repository of data. Unlike actors, they cannot perform any operations. They are used to store data and retrieve the stored data. They represent a data structure, a disk file, or a table in a database.

Representation in DFD – A data store is represented by two parallel lines containing the name of the data store. Each data store is connected to at least one process. Input arrows contain information to modify the contents of the data store, while output arrows contain information retrieved from the data store. When a part of the information is to be retrieved, the output arrow is labelled. An unlabeled arrow denotes full data retrieval. A two-way arrow implies both retrieval and update.

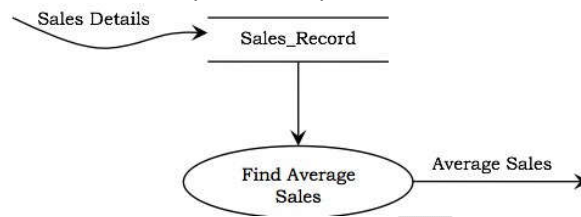


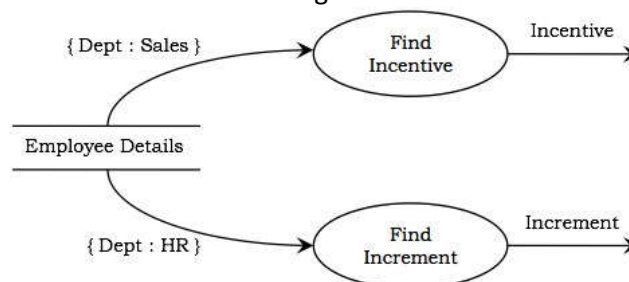
Figure 10: The figure shows a data store, Sales_Record, that stores the details of all sales. Input to the data store comprises of details of sales such as item, billing amount, date, etc. To find the average sales, the process retrieves the sales records and computes the average.

Constraints

Constraints specify the conditions or restrictions that need to be satisfied over time. They allow adding new rules or modifying existing ones. Constraints can appear in all the three models of object-oriented analysis.

- In Object Modelling, the constraints define the relationship between objects. They may also define the relationship between the different values that an object may take at different times.
- In Dynamic Modelling, the constraints define the relationship between the states and events of different objects.
- In Functional Modelling, the constraints define the restrictions on the transformations and computations.

Representation – A constraint is rendered as a string within braces.



Example – The following figure shows a portion of DFD for computing the salary of employees of a company that has decided to give incentives to all employees of the sales department and increment the salary of all employees of the HR department. It can be seen that the constraint {Dept:Sales} causes incentive to be calculated only if the department is sales and the constraint {Dept:HR} causes increment to be computed only if the department is HR.

Control Flows

A process may be associated with a certain Boolean value and is evaluated only if the value is true, though it is not a direct input to the process. These Boolean values are called the control flows.

Representation in DFD – Control flows are represented by a dotted arc from the process producing the Boolean value to the process controlled by them.

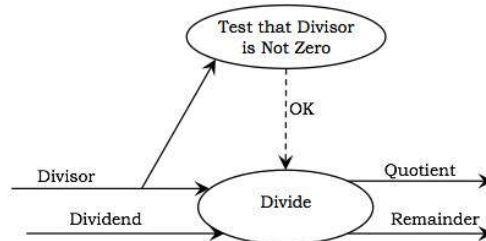


Figure 11: The figure represents a DFD for arithmetic division. The Divisor is tested for non-zero. If it is not zero, the control flow OK has a value True and subsequently the Divide process computes the Quotient and the Remainder.

Developing the DFD Model of a System

In order to develop the DFD model of a system, a hierarchy of DFDs are constructed. The top-level DFD comprises of a single process and the actors interacting with it.

At each successive lower level, further details are gradually included. A process is decomposed into sub-processes, the data flows among the sub-processes are identified, the control flows are determined, and the data stores are defined. While decomposing a process, the data flow into or out of the process should match the data flow at the next level of DFD.

Example – Let us consider a software system, Wholesaler Software, that automates the transactions of a wholesale shop. The shop sells in bulks and has a clientele comprising of merchants and retail shop owners. Each customer is asked to register with his/her particulars and is given a unique customer code, C_Code. Once a sale is done, the shop registers its details and sends the goods for dispatch. Each year, the shop distributes Christmas gifts to its customers, which comprise of a silver coin or a gold coin depending upon the total sales and the decision of the proprietor.

The actors in the system are –

- Customers
- Salesperson
- Proprietor

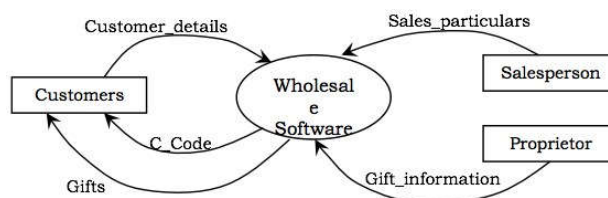


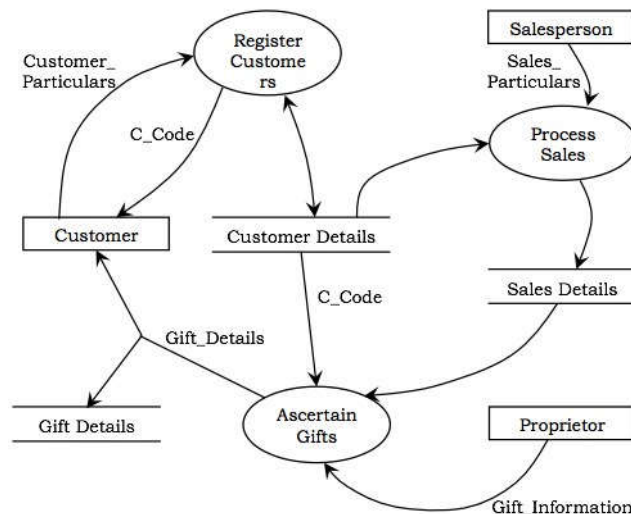
Figure 12: The functional model for the Wholesale Software is given above. The figure below shows the top-level DFD. It shows the software as a single process and the actors that interact with it.

In the system, three processes can be identified, which are –

- Register Customers
- Process Sales
- Ascertain Gifts

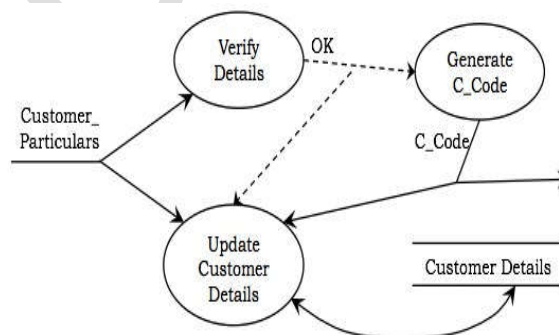
The data stores that will be required are –

- Customer Details
- Sales Details
- Gift Details



DFD of Wholesale Software

Figure 13: In the next level DFD, as shown in the figure, the major processes of the system are identified, the data stores are defined and the interaction of the processes with the actors, and the data stores are established.



DFD of Customer Process

Figure 14: The figure shows the details of the process Register Customer. There are three processes in it, Verify Details, Generate C_Code, and Update Customer Details. When the details of the customer are entered, they are verified. If the data is correct, C_Code is generated and the data store Customer Details is updated.

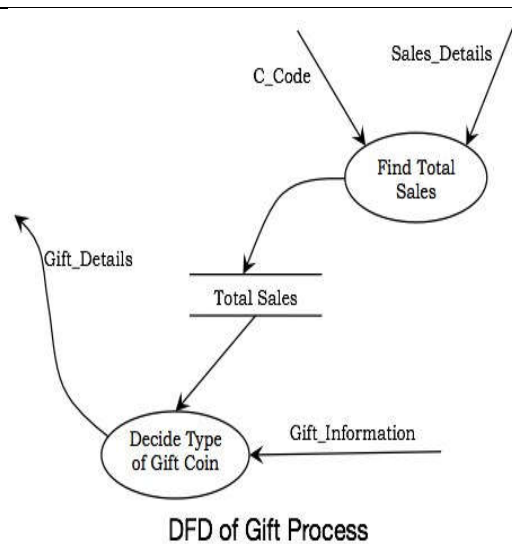


Figure 15: The figure shows the expansion of the process Ascertain Gifts. It has two processes in it, Find Total Sales and Decide Type of Gift Coin. The Find Total Sales process computes the yearly total sales corresponding to each customer and records the data. Taking this record and the decision of the proprietor as inputs, the gift coins are allotted through Decide Type of Gift Coin process.

Advantages and Disadvantages of DFD

Advantages	Disadvantages
DFDs depict the boundaries of a system and hence are helpful in portraying the relationship between the external objects and the processes within the system.	DFDs take a long time to create, which may not be feasible for practical purposes.
They help the users to have a knowledge about the system.	DFDs do not provide any information about the time-dependent behavior, i.e., they do not specify when the transformations are done.
The graphical representation serves as a blueprint for the programmers to develop a system.	They do not throw any light on the frequency of computations or the reasons for computations.
DFDs provide detailed information about the system processes.	The preparation of DFDs is a complex process that needs considerable expertise. Also, it is difficult for a non-technical person to understand.
They are used as a part of the system documentation.	The method of preparation is subjective and leaves ample scope to be imprecise.

Relationship between Object, Dynamic, and Functional Models

The Object Model, the Dynamic Model, and the Functional Model are complementary to each other for a complete Object-Oriented Analysis.

- Object modelling develops the static structure of the software system in terms of objects. Thus, it shows the “doers” of a system.
- Dynamic Modelling develops the temporal behavior of the objects in response to external events. It shows the sequences of operations performed on the objects.
- Functional model gives an overview of what the system should do.

Functional Model and Object Model

The four main parts of a Functional Model in terms of object model are –

- Process – Processes imply the methods of the objects that need to be implemented.
- Actors – Actors are the objects in the object model.
- Data Stores – These are either objects in the object model or attributes of objects.
- Data Flows – Data flows to or from actors represent operations on or by objects. Data flows to or from data stores represent queries or updates.

Functional Model and Dynamic Model

The dynamic model states when the operations are performed, while the functional model states how they are performed and which arguments are needed. As actors are active objects, the dynamic model has to specify when it acts. The data stores are passive objects and they only respond to updates and queries; therefore, the dynamic model need not specify when they act.

Object Model and Dynamic Model

The dynamic model shows the status of the objects and the operations performed on the occurrences of events and the subsequent changes in states. The state of the object as a result of the changes is shown in the object model.



Read Additional Resources

1. **Object-Oriented Paradigm**
Week1-2_Object-Oriented Paradigm.pdf
2. **Object-Oriented Analysis and Design**
Week1-2_Object-Oriented Analysis and Design.pdf
3. **Object-Oriented Analysis and Design**
Week1-2_Object-Oriented Analysis and Design2.pdf
4. **Data Flow Diagram**
Week1-2_Data Flow Diagram.pdf



Watch Video Resources

1. Object-Oriented Paradigm

Week1-2_Object-Oriented Paradigm.mp4

2. Object-Oriented Analysis and Design

Week1-2_Object-Oriented Analysis and Design.mp4

3. Data Flow Diagram

Week1-2_Data Flow Diagram.mp4



Laboratory Activity

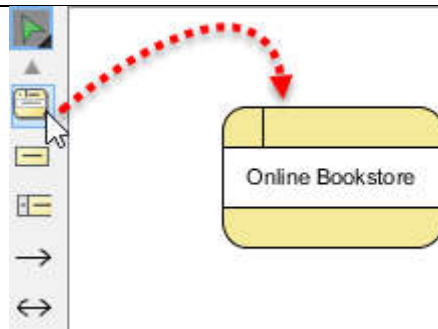
Laboratory Activity 1 -2.1

Data Flow Diagram Comprehensive Guide with Examples

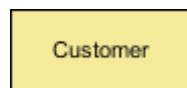
- Creating Context Level DFD Using **Visual Paradigm Online**
- Sign-up for FREE and login to your newly created account @:
<https://online.visual-paradigm.com/diagrams/solutions/free-visual-paradigm-online/>
- or download the Free Community Edition @:
<https://www.visual-paradigm.com/download/community.jsp>

Context level DFD, also known as level 0 DFD, sees the whole system as a single process and emphasis the interaction between the system and external entities. Let's draw a context DFD.

1. To create a DFD, select Diagram > New from the toolbar.
2. In the New Diagram window, select *Data Flow Diagram* and click Next.
3. Enter *Context Diagram* as diagram name and click OK to confirm.
4. Name the diagram *Context Diagram*.
5. We need to create the main process. Drag *Process* from diagram toolbar to diagram. Name it *Online Bookstore*. We will use online bookstore as an example to show you how to create multiple levels DFD.



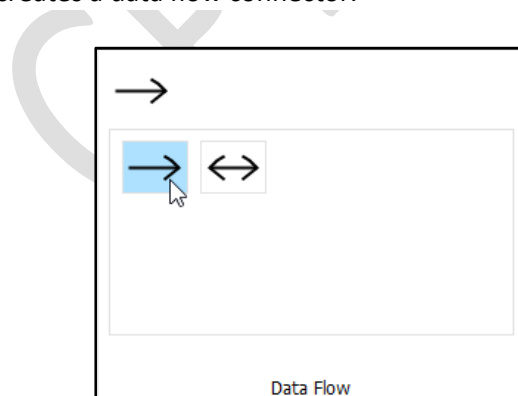
1. As you can imagine, customers will be ordering books through the system. Therefore, add an External Entity to diagram and name it *Customer*. Like creating process, you can create an external entity by dragging External Entity from diagram toolbar to diagram.



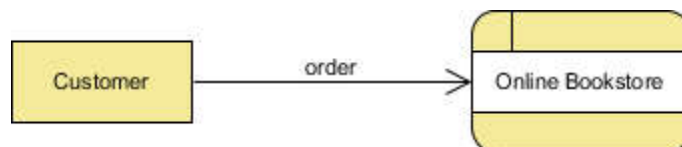
1. A data flow is used to represent the movement of data between different parts of the system. In this case, "order" is a data created by customer and submitted into the system. Therefore, create a data flow from *Customer* to *Online Bookstore*. To do this, move the mouse pointer over *Customer*. Drag out the Resource Catalog button.



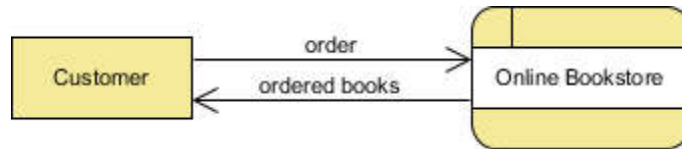
1. Release the mouse button on *Online Bookstore* and select Data Flow from Resource Catalog. This creates a data flow connector.



1. Name it *order*.



1. The book store will finally deliver the books to the customer. Make use of the Resource Catalog to create a data flow from *Online Bookstore* to *Customer*. Name the flow *ordered books*.

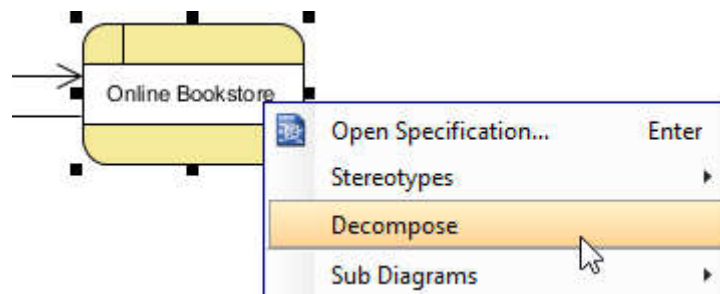


1. That's all for the context diagram. As we see the entire system as a single process, there is just one process shape created in this DFD level. In the next section you will decompose the *Online Bookstore* process and draw its detail.

Creating Level 1 DFD

In this section, you are going to draw the level 1 DFD to show the main functions of online bookstore.

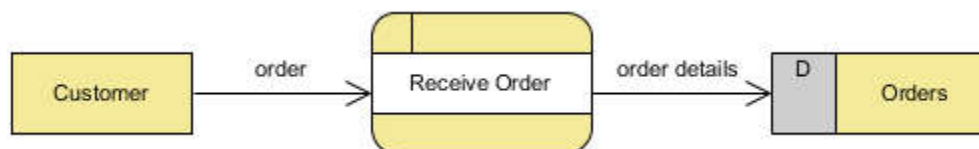
1. Right click on *Online Bookstore* and select Decompose from the popup menu.



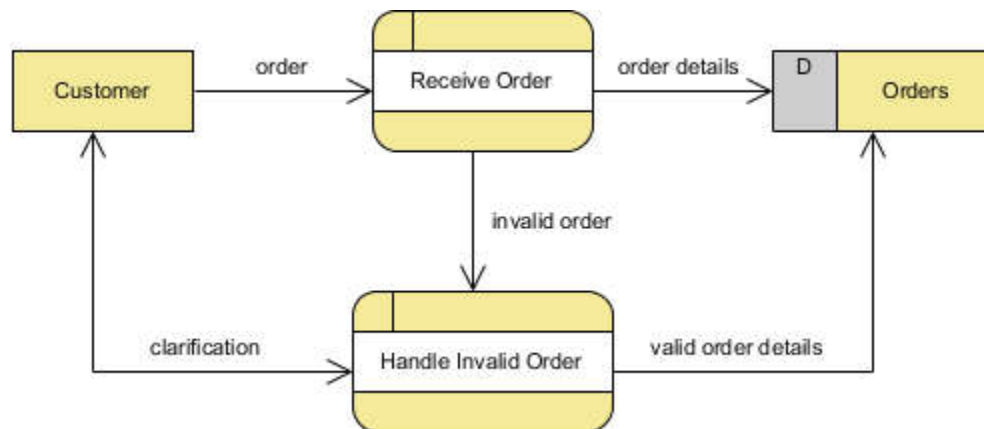
2. The data stores and/or external entities connected to the selected process would be referred to in the level 1 DFD. So, when you are prompted to add them to the new diagram, click Yes to confirm. Note that the new DFD should look very similar to the context diagram initially. Every element should remain unchanged, except that the main process (from which this new DFD decomposes) is now gone and replaced by a blank space (to be elaborated).

3. Right click on the background of diagram and select Rename... from the popup menu. Name the diagram *Level 1 DFD*.

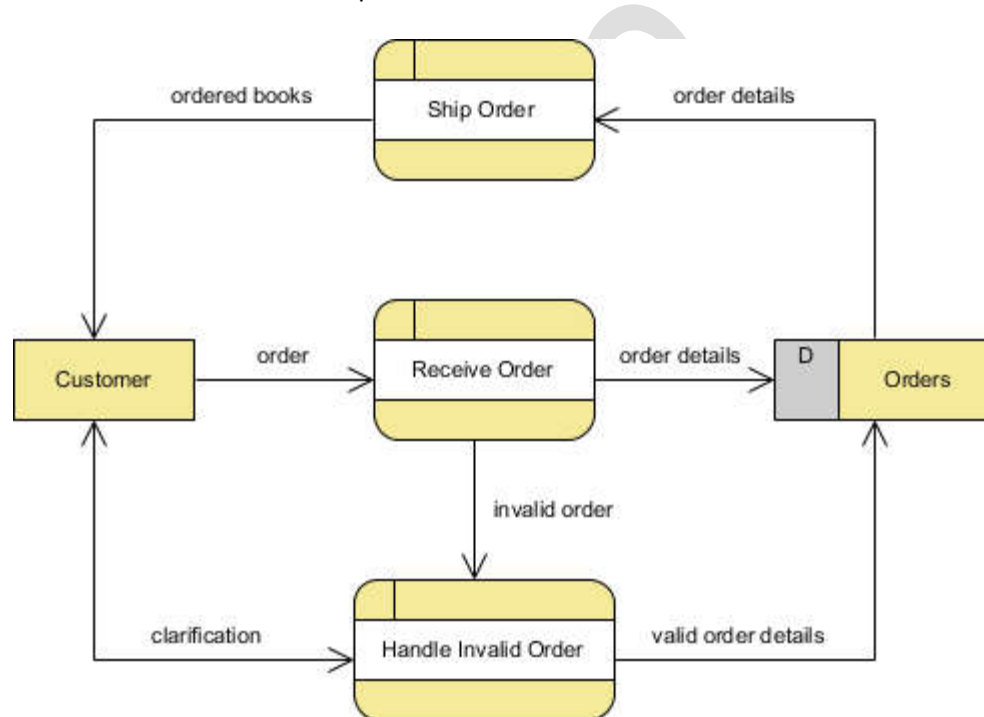
4. There is a process that receives customer order and store it into a database. Let's create a process *Receive Order* and a data store *Orders*. Add data flows in between.



5. When an order is invalid, there should be another process that involves contacting the customer and clarifying his/her need. Create a process *Handle Invalid Order*. Add suitable data flows between the shapes. Here you may use a bi-directional data flow to represent a dialog flow.



6. There is another process that handles shipment. Create the *Ship Order* process. Again, add appropriate data flows between the shapes.



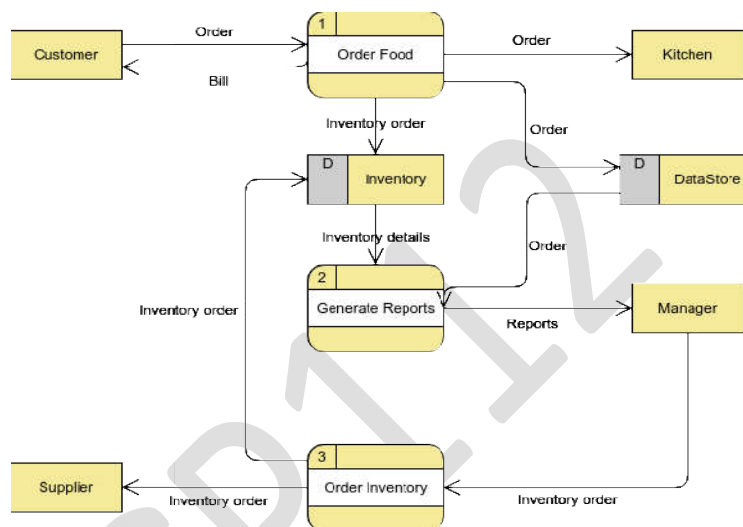
That's all for this tutorial. Depending on the area of interest, you may create further levels to show the system in more detail. As you can imagine, the *Ship Order* process can be elaborated into smaller functional units (processes) like packaging, prepare invoice, arrange deliver, etc.

Activity

Activity 1-2.1

Using Visual Paradigm Online, create the Context level DFD for the Food Ordering System. Screenshot the output and paste it on the Activity Sheet provided in this module and to be submitted on the 3rd Week.

Food Ordering System



Self-Check

Quiz 1-2.1

Instructions: Write your answer on the Answer Sheet (AS) provided in this module.

1. Differentiate object-oriented analysis and object-oriented design (5-points).
2. Enumerate the five parts of a transition. (5-points).
3. Discuss the analysis techniques. (5-points).

Internet References

1. <https://www.youtube.com>
2. <https://www.tutorialspoint.com>
3. <https://warren2lynch.medium.com/data-flow-diagram-comprehensive-guide-with-examples-9858387f25e>
4. <https://www.youtube.com/watch?v=6VGTvgaJlIM>
5. <https://www.youtube.com/watch?v=IKbk4u-ULoQ>