CHAPTER 2: LITERATURE REVIEW OF STRUCTURED AND OBJECT-ORIENTED METHODOLOGIES

2.1 Structured Methodologies

Structured methodologies have been used since the 1970s and have reached maturity. Many systems have been successfully implemented using these methods. However, these methodologies are unable to cope with the increases in user requirements for larger and more complex software systems (Weinberg, Guimaraes and Heath 1990). This chapter will present the historical reasons for the emergence of these methodologies and will describe their tools and techniques that will form the basis for comparison with the Object-Oriented methodologies. The original structured analysis and design (DeMarco and Yourdon and Constantine) will be described. The problems of these methodologies will also be examined.

This history of structured methodologies dates back to the late 1960s when a Systems Development Life Cycle (SDLC) or Classical Project Life Cycle concept was introduced to facilitate the development of large-scale software systems (Fichman and Kemerer 1992; Powers, Cheney and Crow 1990; Thierauf 1986; Whitten, Bentley and Ho 1986; DeMarco 1979; Yourdon 1989). The SDLC is a project management technique which provides developers with some control over the development process, but does not address the quality and productivity of the analysis and design. In the late 1970s. Structured System Development Cycle (SSDC) or Modern Life Cycle concept was introduced to Replace SDLC (Thierauf 1986; DeMarco 1979).

The early emphasis of structured methodologies was mainly function-oriented because batch processing systems were common at that time. This emphasis is seen in the work of Dijkstra (1976), DeMarco (1979), Myers (1975 & 1978), Yourdon and Constantine (1975), Gane and Sarson (1977a & 1977b), and Weinberg (1978). Ward and Mellor (1985) adapted the structured analysis methods to on-line/real-time systems. Recently Yourdon (1989) updated the analysis methods again to what is now known as modern structured analysis. Modern structured analysis
was developed by applying the 'essential' model of McMenamin and Palmer (1984) to structured analysis and more emphasis is placed on data modeling.

2.1.1 Problems with SDLC

Generally, a SDLC breaks down the development process into several distinct project phases. In each project phase, formal documents are produced to be used in the next phase. Different organizations or authors use different SDLCs. Thierauf (1986) describes his SDLC as consisting of six phases: investigation, analysis, design, equipment selection, implementation, and review. Other examples of SDLCs can be found in: Whitten, Hentley and Ho (1986); Powers, Cheney and Crow (1990); DeMarco (1979); Yourdon (1989).

Several problems of SDLC are described by Thierauf (1986), Yourdon (1989), and DeMarco (1979), and they are attributed to the use of bottom-up implementation, the freezing of the specifications, the high maintenance costs, the lack of structured techniques, the failure to meet user requirements, communication problems, and problems with functional specifications.

2.1.2 Bottom-up Implementation Problems

The use of bottom-up implementation required all modules to be tested first, then subsystems, and finally the whole system. This approach may have been derived from the industrial assembly line (Yourdon 1989). This is an effective way of assembling automobiles by testing the prototype model first, but not appropriate for producing one-of-a-kind software systems. There are several problems associated with this approach:

- Nothing is done until the whole project is done (Yourdon 1989; Thierauf 1986). If the deadline is reached during system testing, there will be nothing for the user to see other than the program listings that are of no use to the user.
• The unimportant bugs are discovered at the initial stages of testing, and the important bugs are discovered at the final stages of testing (Yourdon 1989; Thierauf 1986). Simple logic errors are detected inside individual modules during module testing, but interface errors between subsystems are detected during system testing. These interface errors detected at the final stages of a project may cause the recoding of numerous modules.

• Debugging becomes increasingly difficult during system testing (Yourdon 1989). During system testing, all the modules are combined to form the system for the first time. Therefore, it is hard to determine which bug is in which module, especially for large systems that may contain up to hundreds or even thousands of modules.

• During the last stages of testing the need for computer time rises exponentially (Yourdon 1989). A large amount of computer time, possibly twelve hours per day, is required for system testing. Due to the difficulty in obtaining this amount of time, the project tends to fall behind schedule.

2.1.3 Problems with Freezing of Specifications

SDLC insists that one phase must be finished before the next begins (Thierauf 1986; Yourdon 1989). This means that specifications cannot be modified. This is known as freezing the specifications (or design documents). This is unrealistic. Mistakes made by anyone involved in the development of a system specification will result in a flawed product. User requirements may change due to the changing environment such as economics, or government regulations. This approach does not care for these changes.
2.1.4 High Maintenance Costs
The unreliability and inefficiency of the SDLC have led to high maintenance costs (Thierauf 1986). Maintenance has often caused major disturbances to the system. The effects a changed function can have on the whole system cannot be determined except by extensive verification. This is a "hit-or-miss" approach that needs more time than is normally available.

2.1.5 Lack of Structured Techniques
The use of structured techniques tends to be neglected in SDLC (Thierauf 1986; Yourdon 1989). Project leaders, analysts, and programmers’ feel that SDLC is imposed by top-level management. Therefore, they do not feel they need to use structured techniques in SDLC unless management says anything about applying these techniques. Though structured techniques are neglected, they still can be applied in SDLC.

2.1.6 Failure to Meet User Requirements
The SDLC cannot meet the user’s requirements due to the long period needed to produce useable software and insufficient involvement of users (Thierauf 1986). Numerous human errors may occur because of the labor-intensive nature of the SDLC. A large backlog of software systems piles up waiting to be developed because of the inefficiencies of SDLC. Because of the slowness in producing useable software, users consider their involvement a waste of time and tend to see their software departments as a hindrance to progress. This results in increasing dissatisfaction with the use of the SDLC. This approach is not considered to be "user friendly".
2.1.7 Communication Problems

DeMarco (1979) identifies four main factors related to communication problems during systems analysis:

1. Difficulty in describing procedures. Specifications are used for describing procedures. Demonstrating procedures is far easier than describing them.

2. Inappropriate method (narrative test) used by analysts. Most analysts prefer using a thousand words of narrative text than a picture for writing specifications. Users are interested in the system interface and procedures. However, these tend to be neglected until after the completion of analysis and the production of user manuals.

3. Different languages used by analysts and users. Users are interested in the human interface (or human procedures) to the system that is left out by the analysts until after analysis when the user manuals are completed. Analysts work with specifications, flowcharts, data format descriptions, disk and core maps, and code that are not appropriate for most users.

4. There is no early working model of the system. Systems described by the analysts exist in only the analysts' minds. There is no useable model for them.

2.1.8 Problems with Functional Specifications

Several problems with the functional specifications are described by DeMarco (1979). The analysis is more complex for larger systems and therefore results in larger functional specifications. These specifications are written in "Victorian novel" style that cannot be understood unless they are read from beginning to end. Further problems with these specifications include: excessive redundancy, excessively wordy, excessively physical, and tedious to read and write.
2.1.9 Purpose and Goals of Structured Methodologies
Structured methodologies cannot completely solve the problems of SDLC, but can help to reduce their magnitude by providing a more productive and disciplined environment (DeMarco 1979; Thierauf 1986). Communication problems between analysts and users are tackled by replacing text with graphics. The problem with the freezing of the specifications is dealt with by replacing functional specifications with the more maintainable structured specifications. Structured specifications are partitioned into smaller, readable, and selectable mini-specifications. Narrative text is replaced with graphical equivalence in structured specifications and redundancy and physical information are removed. Bottom-up implementation is replaced by top-down implementation (DeMarco 1979; Yourdon 1989).

The SDLC was modified, giving rise to the Structured System Development Cycle (SSDC). Former analysis and design phases are replaced by structured analysis and design phases and the previous implementation phase is replaced by top-down implementation phase. For example, DeMarco (1979) describes his SSDC as consisting of five phases: survey, structured analysis, structured design, hardware study, and top-down implementation.

2.1.10 Structured Paradigm
Different structured tools and techniques are designed for different purposes. In this section these tools and techniques are described with simple examples to illustrate how they are used.

2.1.10.1 Data Flow Diagram (DFD)
The data flow diagram (DFD) is one of the main tools of structured analysis. It is one way of viewing a system, namely the function-oriented view. It places more emphasis on functions than data or behavior of the system. DFDs are used to enhance communication between analysts and users because they are simple and easily understood by users (Powers, Cheney and Crow 1990). DFDs consist of four kinds of symbol that represents four kinds of system components, namely, external entities, data stores, data flows and processes.
2.1.10.2 The External Entity

An external entity is a person, a department, an organization or another system outside the system being modeled. External entities communicate with the system being modeled by either supplying data to the system or receiving data from the system. External entities may be a client, an accounting department, a bank, or an inventory system. External entities are sometimes called sources/sinks (DeMarco 1979), terminators (Yourdon 1989), origins/destinations (Teague and Pidgeon 1985). Figure 2.1 shows the symbols used for the external entities.

(Gane and Sarsons 1979) (Yourdon 1979) Example (Yourdon)

![Figure 2.1: Notations for External Entities](image)

2.1.10.3 The Data Store

A data store is a placed where data is deposited to or retrieved from. It may be a file, an area of disk or tape, a database, a collection of paper records such as time cards or invoices. Data stores show data at rest. Data stores are often called files (DeMarco 1979). Figure 2.2 shows the notations used for the data stores.

(Gane and Sarsons 1979) (Yourdon 1979) Example (Gane and Sarson)

![Figure 2.2: Notations for Data Stores](image)
2.1.10.4 The Data Flow

The data flow is a movement of packets of data from one point of the system to another. Packets of data may be a printed report, a computer input, a form, data stored in or retrieved from a file. Figure 2.3 shows the symbols used for the data flows.

(Gane and Sarsons 1979)  (Yourdon 1979)  Example

DATA FLOW  DATA FLOW  TIME-SHEET

Figure 2.3: Notations for Data Flows

2.1.10.5 The process

A process is a point within a system where incoming data flows are changed into outgoing data flows. Processes may be work that is done by computers, machines or people. Processes are sometimes called bubbles or circles (DeMarco 1979), transformation or transform (Teague and Pidgeon 1985), or function (Yourdon 1989). Figure 2.4 shows symbols used for the processes.

(Gane and Sarsons 1979)  (Yourdon 1979)  Example (Yourdon)

PROCESS NO.

PROCESS NAME

PROCESS NO.

PROCESS NAME

1.

PRODUCE PAYROLL

Figure 2.4: Notation for Processes
2.1.11 Data Dictionary

A data dictionary is a catalogue of all the data elements within the system. Each data element is precisely defined so that both analysts and user will have a common understanding of the term being used. The definitions of the data stores and data flows are expanded in the data dictionary. An example of an entry in a data dictionary is shown in Figure 2.5:

Employee-Data = Employee-Name + Employee-Address + Employee-Number + Employee-Salary

Figure 2.5: An example of an Entry in a Data Dictionary

The "=" sign in the above example means "is composed of" and the "+" sign means "and" (Yourdon 1989).

2.1.12 Process Specifications

Process specification is used to describe in detail what is actually happening inside each bottom-level process (circle or bubble) of a data flow diagram. It is also known as minispec, that is, miniature specification (DeMarco 1979; Gane and Sarson 1977a and 1977b; Weinberg 1978). There are several tools that can be used for process specification: structured English, Decision tables, decision trees, pre/post conditions, flowcharts (Chapin 1970), narrative English, Nassi-Shneiderman diagrams (Nassi and Shneiderman 1973), Ferstl diagrams (Ferstl 1978), Hamilton-Zeldin diagrams (Hamilton and Zeldin 1972), and problem analysis diagrams (Futamura et al. 1981). Only three popular tools are described here: structured English, decision tables and decision trees.
2.1.12.1 **Structured English**

Structured English is based on structured programming and is a tool for describing processes. Structured English uses the three basic constructs: sequence, selection and iteration. This technique restricts the vocabulary to action verbs and terms that are in the data dictionary. Structured English is easily understood by the user and allows precise statements of rules. Figure 2.6 shows an example of Structured English that is taken from DeMarco (1979).

**POLICY FOR INVOICE PROCESSING**

If the amount of the invoice exceed $500,

If the account has any invoice more than 60 days overdue,

hold the confirmation pending resolution of the debt.

Else (account is in good standing)

issue confirmation and invoice.

Else (invoice $500 or less)

If the account has any invoice more than 60 days overdue,

issue confirmation, invoice and write message on the credit action report.

Else (account is in good standing)

issue confirmation and invoice.

*Figure 2.6: An example of Structured English*

(Source: DeMarco 1979:43)
2.1.12.2 Decision Tables
Decision tables are an alternative tool used for process description. Decision tables consist of three components: conditions that described all the conditions (or factors) applied to the data that will affect the decision; actions that describe the possible actions that can be taken; rules that describe what actions are to be taken under what conditions. The policy invoice processing example given under the Structured English section can be described by a decision table as shown in Figure 2.7.

<table>
<thead>
<tr>
<th>CONDITIONS</th>
<th>RULES</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Invoice &gt; $500</td>
<td>1 2   3 4</td>
</tr>
<tr>
<td>2. Account over-due by 60+days</td>
<td>Y N Y N</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ACTIONS</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Issue Confirmation</td>
<td>N Y Y Y</td>
</tr>
<tr>
<td>2. Issue Invoice</td>
<td>N Y Y Y</td>
</tr>
<tr>
<td>3. Msg to C.A.R.</td>
<td>N Y N N</td>
</tr>
</tbody>
</table>

*Figure 2.7: An example of Decision Table (Source: DeMarco 1979:44)*

2.1.12.3 Decision Tree
The decision tree is an alternative tool for process description. The use of decision trees is more effective when there is one action for each rule. A decision tree is a horizontal tree. The root of the tree is shown at the left and represents the first decision. The nodes of the tree represent decision points. The branches of each node represent the possible outcomes of that decision point. Actions are represented by the leaves of the tree. Each path starting with the root ending with a leaf representing a decision rule. Again the policy invoice processing example can be done using the decision tree and is illustrated in Figure 2.8.
2.1.13 Entity-Relationship Diagram (ERD)

Although the data stores are depicted in the data flow diagram, the details of the data are purposely omitted because the main emphasis of the data flow diagrams is on the functions and not the data of the system. Both the details of the data and the relationships between the data stores are absent from the data flow diagrams. To model the stored data of the system and their relationships, another modeling tool is used, namely, the entity-relationship diagram (ERD) (Flavin 1981; Chen 1976; Martin 1982; Date 1986). There are at least three other notations for ERD such as Bachman diagrams (Beck 1989), DeMarco’s data structure diagrams (DeMarco 1979), and Jackson’s data structure diagrams (1975). The entity-relationship diagram consists of three main components: entity sets, relationship sets and attributes (Hawryszkiewycz 1988).
2.1.13.1 Entity sets
An entity set or an object type (Yourdon 1989) is a group of objects (things) having common properties. For example, an EMPLOYEE entity set is a group of people that can be represented by their names, addresses, employee numbers and salaries. Figure 2.9 shows an entity set as a rectangular box on the entity-relationship diagram.

![Figure 2.9: An Entity Set](image)

2.1.13.2 Relationship sets
A relationship set represents a set of relationships between entities in two different entity sets. Figure 2.10 shows a relationship set as a diamond-shaped box on the entity-relationship diagram.

![Figure 2.10: A relationship set](image)
2.1.13.3 Attributes
Attributes are properties of entities in an entity set. Figure 2.11 shows one way of representing attributes on the entity-relationship diagram.

![Diagram](image)

Figure 2.11: Attributes

2.1.14 State-Transition Diagram (STD)
Real-time systems and online systems are dependent on the time for data to be accessed and functions to be performed. These time-dependent behaviors of the system must be modeled as well as functions and data of the system. The modeling tool used for the time-dependent behavior of the system is the state-transition diagram (STD). The state-transition diagram consists of two main components: states and state-changes (Yourdon 1989).
2.1.14.1 States
A state that a system is in represents a period of time in which a certain behavior of the system is observable. Figure 2.12 depicts a state as a rectangle box.

![IDLE and WAITING states](image)

*Figure 2.12: States*

2.1.14.2 State-Changes
A state-change represents a transition from one state to another. A state-change is accompanied by the *conditions* that cause the change of state and the *actions* to be taken when a change of state occurs. Figure 2.13 shows the state-changes as arrows.
2.1.15 Structure Chart

A structure chart is a modeling tool used during the design phase to represent the system architecture (a hierarchy of modules) (Yourdon 1989). Another diagramming tool that can be used in place of structure chart is HIPO diagrams (IBM 1974). A structure chart consists of three main components: the module, the connection and the couple (DeMarco 1979). Figure 2.14 shows these major components.
Figure 2.14: Components of a Structured Chart
(Adapted from: Yourdon 1989)

The module is depicted by a rectangular box on the structure chart. Modules represent subroutine, procedures, or subprograms. The connection is depicted by the arrow connecting two modules. The connections represent a module being called by another. The couple is depicted by a short arrow. The couple represents parameters passing between modules.
2.1.16 Top-down Decomposition

DFD needs to be simple in order for it to be an effective communication tool. In practice even a medium-sized system may need as many as two hundred processes for it to be properly represented in a single DFD (Powers, Cheney and Crow 1990). It would be very difficult to follow and understand such a DFD with too many processes on it. To simplify the DFD, structured methodologies apply a technique called Top-down Decomposition (DeMarco 1979), Hierarchical (Top-down) Partitioning (Powers, Cheney and Crow 1990), or Levelling (Yourdon 1989; Hawryszkiewycz 1988). This technique first represents a system using a simple top-level DFD consisting of only one process with data flows flowing from this process to external entities. This DFD is called the Context Diagram. Figure 2.15 shows that the process in the context diagram is decomposed into a lower level DFD called Diagram 0 and each process in Diagram 0 is further decomposed into another lower level DFD of increasing detail. Decomposition continues until all the processes cannot be decomposed further. The lowest level process or the unpartitioned process is called a primitive function or primitive process as shown in Diagram 4.5 of Figure 2.15.
Figure 2.15: Hierarchy of DFDs
(Adapted from: Teague and Pidgeon 1985)
2.1.17 Structured Methods

Many structured methods are currently in existence. This section only represents the original Structured Analysis (DeMarco 1979) and Structured Design (Yourdon and Constantine 1975, 1979 and 1989).

2.1.17.1 Structured Analysis/Structured Design (SA/SD)

Structured Analysis/Structured Design is a combination of Structured Analysis (DeMarco 1979 and Structured Design (Yourdon and Constantine 1975, 1979 and 1989). Figure 2.16 shows the SSDC as described by DeMarco (1979).

![Diagram of SSDC]

Figure 2.16: Structured System Development Cycle (SSDC)
(Adapted from: DeMarco 1979)
Figure 2.17 shows the decomposition of Process 2 (structured analysis) of Figure 2.16.

![Diagram of structured analysis process]

*Figure 2.17: Structured Analysis (Adapted from: DeMarco 1979)*

A structured analysis phase consists of seven processes as seen in Figure 2.17. Each of these processes is detailed in DeMarco (1979) and briefly summarized below:

1. Study current environment (Process 2.1). The purpose of this process is to determine how the system works currently. The product is a Current Physical DFD.

2. Derive logical equivalent (Process 2.2). The purpose of this process is to determine what the system does currently. It involves transforming the current physical system into the current logical system. The current physical DFD is used to produce a Current Logical DFD.
3. *Model new logical system (Process 2.3).* The purpose of this process is to determine what the system to be developed will do. The products of this process include a New Logical DFD for showing interface and partitioning, Data Dictionary for documenting data flows and data stores, and transform descriptions for describing the DFD processes in detail.

4. *Establishing the Man-Machine Boundary (Process (2.4).* The purpose of this process is to determine how much of the new system should be automated and how much it should remain mutual. Several new Physical DFDs are produced.

5. *Quantifying each option (Process 2.5).* The purpose of this process is to determine the costs and benefits of each of the new Physical DFDs produced by the previous process. A document, consisting of all the options and their associated costs and benefits, is produced.

6. *Select option (Process 2.6).* This process is done by management not the analyst. The selected DFD is passed to the next process.

7. *Packaging the specification (Process 2.7).* This process involves combining all the components into a structured specification. The structured specification produced consists of DFDs, Data Dictionary and Transform Descriptions.

The structured specification is then passed into the structured design phase to transform the DFDs into structured charts. System design modeling is mainly performed using structured charts.
2.1.17.2 Other Structured Methods

Several other structured methods are summarized in Appendix A. These are Information Engineering (Martin 1990), Structured Systems Analysis and Design Method (Longworth and Nicholls 1986), Structured Analysis and Design Technique (Ross and Schoman 1977), Information Systems Activity and Change Analysis (Lundeberg, Goldkuhl and Nilsson 1981), Jackson System Development (Jackson 1983; Sutcliffe 1988), Nijssen’s Information Analysis Method (Nijssen 1978), and Mascot-3 (Simpson 1986).
2.2: OBJECT-ORIENTED METHODOLOGIES

Many Object-Oriented methodologies have been developed within the past ten years, but currently there is no universal standard. The more mature and better recognized methodologies are that of Booch (Booch 1991), Rumbaugh *et al.* (Rumbaugh *et al.* 1992), and Coad & Yourdon (Coad & Yourdon 1991a) because they cover the whole development cycle.

2.2.1 History of Object-Orientation

Recently, Object-Oriented software system development has attracted the interest of many computing professionals and researchers, and this interest is reflected in computing literature and conferences. However, the history of Object-Orientation can be traced back to the beginning of the 1960s when there was a need to develop tools for the simulation of systems such as 'nerve networks, communications systems, traffic flow, production systems, administration systems, and even social systems' (Martin 1993:261).

The history of Object-Orientation began in 1961, when Kristen Nygaard started designing a simulation language and was later joined by Ole-Johan Dahl at the Norwegian Computer Center and the University of Oslo in Norway to develop the language originally known as *Simula*. As the name suggests, *Simula* was designed for use in simulation modeling, but quickly found its way into prototyping and application development. In 1965, planning started for an improved version of *Simula* to be used for general purpose programming. By the end of 1966, the required groundwork for this new, general purpose programming language, known as *Simula 67*, was completed. In 1986 the name of this language reverted to its original name, *Simula*. 
In the early 1970s, Alan Kay, led a group, including Daniel Ingalls and Adele Goldberg, in the development of another Object-Oriented technology. This group was the Learning Research Group (LRG) of Xerox Research Center in Palo Alto (PARC), California. Their task was to develop the Interim Dynabook Personal Computer in which the language called Smalltalk was used as the software component. According to Kay (1993), Smalltalk was so named because of 'a reaction against the "IndoEuropean god theory" where the systems were named Zeus, Odin, and Thor, and hardly did anything. Smalltalk was so innocuous that if it ever did anything nice people would be pleasantly surprised'. Smalltalk was originally designed to be used as a research tool. In the mid 1980s, Xerox formed ParcPlace Systems, a subsidiary company, to market Smalltalk as a commercial product (Henderson-Sellers 1992a). It was during the development of Smalltalk that the term Object-Oriented was introduced.

The increased in the popularity of the C programming language motivated several people to extend its capabilities to allow Object-Oriented features. Bjarne Stroustrup of AT&T started the development of the C++ language in 1979. According to Stroustrup (1993), 'C++ was designed to provide Simula's facilities for program organization together with C's efficiency and flexibility for systems programming'. C++, as the name suggests, attempted to provide more features than C. While C++ attempted to solve the shortcomings of C, at the same time, it introduced new problems. Several papers described some of these problems including those written by Edelson and Pohl (1989), and Moylan (1992) as well as Joyner (1992).

Objective-C is another language that provides an Object-Oriented extension to the C language. Objective-C was developed by Brad Cox of Stepstone Corporation (Productivity Products International). According to Meyer (1988b), Objective-C is different from C++ in many ways, even though their aims are similar. While C++ has a Simula influence, Objective-C has a Smalltalk influence.
Recently, Bertrand Meyer developed a new language called Eiffel, which was strongly influenced by Simula, and like Smalltalk, is purely Object-Oriented. Therefore, it is impossible to “slip back” to procedural programming, as it is in the hybrid C-extensions’ (Henderson-Sellers 1992a). According to Meyer (1988b), he developed Eiffel because he found no existing language that satisfied his expectations. Eiffel was designed in an attempt to support the concepts of Object-Orientation. Nevertheless, Eiffel has also received some criticism such as performance problems and lack of supporting tools (Graham 1991).

Like Structured methods, Object-Oriented methods have their origin in Programming. As programming languages are maturing, the current phase of interest in Object-Orientation is shifting to design, analysis, and specifications.

2.2.2 Object-Oriented Paradigm
The Object-Oriented paradigm as a whole is not just programming, but involves the whole software development life cycle, i.e., analysis, design, and implementation. The terms which often appear in the Object-Oriented literature are objects, classes, instances, methods, requests, messages, inheritance, encapsulation, abstraction, information hiding, early or static binding, late or dynamic binding, and polymorphism. In this section, these terms are defined and their relationships with one another are explained.
2.2.2.1 Object

An object is any real or abstract entity which encapsulates data and operations that process the encapsulated data. Figure 2.1.1 illustrates object encapsulation.

![Object Encapsulation Diagram](image)

*Figure 2.1.1: Object Encapsulation (Adapted from: Sharble & Cohen 1993)*

Objects may be distinguished as *external* and *internal* objects (Meyer 1988b). Anything that is visible, and has physical existence, such as a person, may be called an *external* object. Anything that is not visible, and has no physical existence, but has computer representation, such as a record, may be called an *internal object*. Any non-object, such as a function that does not appear during the analysis stage, can be made into an internal object during the design and the implementation stage. This process of creating an object out of a non-object, so that it can be manipulated, is called *reification* (Rumbaugh 1992). An object is composed of data and operations. An operation processes some data in an object to cause the object to behave in a certain way. An object may contain other objects. This enable complex objects to be created. Objects are the elementary building blocks in the Object-Oriented paradigm.
2.2.2.2 Object Type
An object type is one kind of object. Objects that have the same data structures and operations are of the same object type. The object attributes are represented by the data structures and the object behaviors are represented by the operations. For example, if PERSON is an object, then EMPLOYEE is one kind of PERSON. ENGINEER is another kind of PERSON (see Figure 2.1.2).

![Object Types Diagram](image)

*Figure 2.1.2: Object Types*

2.2.2.3 Instance
An instance is an actual existence of a particular object type. Each instance is unique. An individual object is an instance of its own type. If John Smith is employed by a certain company, then JOHN-SMITH belongs to a kind of PERSON called EMPLOYEE (see Figure 2.1.3). JOHN-SMITH is an instance of EMPLOYEE, and JOHN-SMITH is a PERSON (i.e., an object).
2.2.2.4 Operation

An operation is a routine within an object that is used to process the data of that object. The data structures of one object type should not be directly accessed by the operations of another object from another object type. A message must be sent from one object to another to access the data structures of that object. Examples of operations may be to Display-Employee-Details or to Change-Employee-Salary (see Figure 2.1.4).

Figure 2.1.4: Operations of an Object Type
2.2.2.5 **Method**

A method is a language implementation of an operation. Smalltalk uses *method*, while C++ uses *member function* to implement an operation.

2.2.2.6 **Encapsulation**

Encapsulation is the act of combining data and its operations together into an object. Encapsulation allows one object to hide its information from another object. This is shown as Information Hiding. Henderson-Seller (1992) differentiates encapsulation from information hiding (see Figure 2.1.5) by referring to encapsulation as packaging code and data together into an object, but the code and the data are still visible to other objects. That is, encapsulation does not necessarily mean information hiding, but information hiding does always mean encapsulation in Object-Orientation. Information hiding protects the data from invalid use and hence from corruption. Users may know what operations to request of an object, but not the implementation details of the operations. This allows the implementations to be changed without changing the applications that call them.
2.2.2.7 Requests

A request is the act of asking an object to perform one of its own operations using the specified parameters. A request must be send to an object to request it to perform a specified operation. A response may optionally be returned after the specified operation has performed the appropriate method. A request contains the object name, the operation name, and the parameters needed by the operation. A request might be:

JOHN-SMITH Change-Employee-Salary NEW-SALARY.

Here JOHN-SMITH is the object name, Change-Employee-Salary is the operation name, and NEW-SALARY is the parameter. Another way of showings request might be:

Change-Employee-Salary (JOHN-SMITH, NEW-SALARY)
By using this notation, multiple recipient objects are allowed and the user does not need to know which is the recipient object and which is the parameter.

2.2.2.8 Messages

A message is defined as a request sent to an object to perform the specified operation using the parameters given and to return the result. Messages are Object-Oriented implementations of requests. In Figure 2.1.6 the arrow us used to represent a message flowing from the sender to the receiver. The receiver performs some kind of operation and then a result is returned to the sender.

![Message Connection Diagram](source: Coad & Yourdon 1991a: 150)

2.2.2.9 Class

A class is an object type that is implemented in a programming language. It contains data structures and methods. The methods specify the operations that can be used with the data structures. During analysis, object types are used. During design and implementation, classes are used. For example, an implementation of an EMPLOYEE object type may be called an EMPLOYEE class. It might contain data structures such as Name, Position, Salary, Location, and methods such as Promote, Change-Salary, and Change-Location. Once a method is stored, it can be shared by all objects of that class.
2.2.2.10 Abstract Data Type (ADT)
An abstract data type (ADT) is a user-defined type (UDT) that describes an object type. An ADT can be thought of as a class without implementation. Conversely, a class can be thought of as an ADT with implementation.

2.2.2.11 Inheritance
Inheritance is a mechanism for creating new classes (subclasses) from existing classes (superclasses) by allowing the new classes to inherit the data structures and methods of the existing classes. An object subclass may be specialized by adding new data structures and methods or by changing the inherited data structures and methods (Weinberg, Guimaraes, and Heath 1990). A subclass can be created from two or more superclasses. This is called multiple inheritance. A subclass created from only one superclass, may be called single (hierarchical) inheritance. For Example, EMPLOYEE, a superclass, may be subclasses such as MANAGER, SECRETARY, and EXECUTIVE, each with inherited, as well as their own data structures and operations (see Figure 2.1.7).

![Inheritance Diagram](image)

*Figure 2.1.7: Inheritances (represented by Gen-Spec notations)*
2.2.2.12 Polymorphism

Polymorphism is the ability to accept the same request by two or more different classes, but each with its own individual response. Polymorphism originated from *poly* which means *many*, and *morph* which means *form*. Therefore, polymorphism means many forms. For example, the EMPLOYEE object type contains the operation Display-Employee-Details. The EXECUTIVE object subtype inherits this operation from EMPLOYEE object type automatically. During the implementation, the EMPLOYEE class has the method for Display-Employee-Details operation. This method is inherited by EXECUTIVE subclass. However, a company may choose to display the executive details differently from the normal employee details. So a new or modified method for a Display-Employee-Details operation must be implemented for EXECUTIVE class. The new or modified method for the EXECUTIVE replaces the original method inherited from EMPLOYEE class. Although the method of EXECUTIVE subclass for the Display-Employee-Details operation is different from that of the EMPLOYEE, the Display-Employee-Details operation is still unchanged (see Figure 2.1.8). For example, the JOHN-SMITH object that belongs to the EXECUTIVE subclass would respond to the message Display-Employee-Details(JOHN-SMITH) differently from the JOHN-SMITH object that belongs to the EMPLOYEE class. Polymorphism allows a request to be made without knowing the method invoked. The selection mechanism in the Object-Oriented implementation takes care of this.
2.2.13 Binding

Binding is the process of choosing the appropriate method based on the object type. Table 2.1.1 depicts two categories of binding:

<table>
<thead>
<tr>
<th>Binding done when the program is compiled is called</th>
<th>Binding done when the program is run is called</th>
</tr>
</thead>
<tbody>
<tr>
<td>• compile-time binding,</td>
<td>• run-time binding,</td>
</tr>
<tr>
<td>• early binding, or</td>
<td>• late binding, or</td>
</tr>
<tr>
<td>• static binding.</td>
<td>• dynamic binding.</td>
</tr>
</tbody>
</table>

*TABLE 2.1.1: DIFFERENT TERMINOLOGIES USED FOR BINDING  
(Adapted from: Martin 1993:275)*
Polymorphism is made possible through dynamic binding. Dynamic binding is important in Object-Ontention because an object type to which the method belongs can be determined at run-time. For example, a method for EMPLOYEE class may be to Print-Employee-Details to a printer. Which printer to print to may not be known until run-time. Therefore, the method used may not be known until the user selects the printer. Dynamic binding causes overheads during run-time, but it adds flexibility and maintainability to an application.

**2.2.2.14 Link**

A link is a connection between instances (objects). For example:

JOHN-SMITH Works-for Company-X.

**2.2.2.15 State**

A state of an object is the form in which the object is in. The value of the attributes of the object express the state of that object. For example, the TRAFFIC-LIGHT object may be in any one of the three states: GREEN, YELLOW, and RED. Each of these states can be stored as a value in the attribute called COLOUR (see Figure 2.1.9).

![Figure 2.1.9: State of TRAFFIC-LIGHT Object](image-url)
2.2.2.16 Events
An event is a happening at some point in time. An event occurs when there is a change of state of an object. Events trigger operations and operations in turn cause events to occur. Events can occur in one direction only and do not return a value. An object responds to an event by changing its state, sending an event to the sending object, or sending an event to a third object. For example, an event occurs when a user presses down the right button of a mouse.

2.2.2.17 Association
An association is a kind of link. An association describes a category of link the same as an object type describes a category of object. For example, PERSON Works-for-COMPANY. In this example, Works-for is an association. Figure 2.1.10 shows an association using graphic notation.

![Figure 2.1.10: Association or Instance Connection](image)

2.2.2.18 Aggregation
Aggregation is a type of association, which describes the relationship between the parts and whole. Aggregation is the consists-of and the a-part-of relationship. For example, a house consists-of three bedrooms, a lounge, a kitchen, and a bathroom (see Figure 2.1.11). It is equally valid to say that a lounge is a-part-of a house.
2.2.2.19 Generalization

Generalization is defined as the relationship between a superclass and its subclasses. Each subclass inherits the properties of its superclass. Each instance of a subclass is also an instance of its superclass. Generalization is the is-a relationship. Rumbaugh, J. et al. (1991:42) define inheritance as 'the mechanism of sharing attributes and operations using the generalization relationship'. For example, MANAGER is an employee. In this case MANAGER is a subclass and EMPLOYEE is a superclass. MANAGER inherits the properties of EMPLOYEE. If John Smith is a manager, John Smith is also an employee, i.e., if JOHN-SMITH is an instance of MANAGER, then JOHN-SMITH is also an instance of EMPLOYEE (see Figure 2.1.12).
2.2.2.20 Terminologies Used by Language Implementation

Different terminologies are used in different language implementations of the Object-Oriented paradigm. Table 2.1.2 compares the terminology used by six well-known languages:

TABLE 2.1.2: LANGUAGE COMPARISON OF OBJECT-ORIENTED TERMINOLOGY
(Adapted from: Henderson-Sellers 1992:264)

<table>
<thead>
<tr>
<th>Smalltalk</th>
<th>C++</th>
<th>Objective-C</th>
<th>Object-Pascal</th>
<th>Eiffel</th>
<th>CLOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object</td>
<td>Object</td>
<td>Object</td>
<td>Object</td>
<td>Object</td>
<td>Instance</td>
</tr>
<tr>
<td>Class</td>
<td>Class</td>
<td>Factory</td>
<td>Object type</td>
<td>Class</td>
<td>Class</td>
</tr>
<tr>
<td>Method</td>
<td>Member function</td>
<td>Method</td>
<td>Method</td>
<td>Routine</td>
<td>Method Generic function</td>
</tr>
<tr>
<td>Instance variable</td>
<td>Member</td>
<td>Instance variable</td>
<td>Object variable</td>
<td>Attribute</td>
<td>Slots</td>
</tr>
<tr>
<td>Message</td>
<td>Function call</td>
<td>Message expression</td>
<td>Message</td>
<td>Applying a routine</td>
<td>Generic function</td>
</tr>
<tr>
<td>Subclass</td>
<td>Derived class</td>
<td>Subclass</td>
<td>Descendent type</td>
<td>Descendent</td>
<td>Subclass</td>
</tr>
<tr>
<td>Inheritance</td>
<td>Derivation</td>
<td>Inheritance</td>
<td>Inheritance</td>
<td>Inheritance</td>
<td>Inheritance</td>
</tr>
</tbody>
</table>
2.2.3 Object-Oriented Methods

Mrdalj (1990) provides 83 references of Object-Oriented development which shows the amount of research in this area. In this section Coad & Yourdon OOA and OOD are summarized.

2.2.3.1 Coad & Yourdon OOA & OOD

The Object-Oriented Analysis (OOA)/Object-Oriented Design (OOD) Model called “The Multi-Layer, Multi-Component Model” by Coad & Yourdon (1991a and 1991b) consists of four main components: Human Interaction, Problem Domain, Task Management, and Data Management. Within each of these components there are five layers: Subject, Class-&-Object, Structure, Attribute, and Service. Figure 2.1.13 shows this model graphically.

![Multi-Layer, Multi-Component Model](image)

*Figure 2.1.13: Multi-Layer, Multi-Component Model (Adapted from: Coad & Yourdon 1991a:179)*

During OOA, only the Problem Domain Component is modeled. During OOD, the Problem Domain component is improved and the other three components are modeled.
2.2.3.1.1 Four Components, Four Activities

There are four activities corresponding to the four components shown in Figure 2.1.13. These activities are: Designing the Human Interaction Component, Designing the Problem Domain Component, Designing the Task Management Component, and Designing the Data Management Component. What is involved in each of these activities is described below:

- **Human Interaction Component (HIC)/Designing the Human Interaction Component.** The HIC component may contain the actual inputs and outputs. Examples of classes for HIC include Selector, Pane, and Window.

- **Problem Domain Component (PDC)/Designing the Problem Domain Component.** This activity involves improving OOA. Examples of classes for PDC include Owner, Clerk, Vehicle, and LegalEvent.

- **Task Management Component (TMC)/ Designing the Task Management Component.** The TMC may contain communication, coordination, and real time task definition. Examples of classes include Task and Task Manager.

- **Data Management Component (DMC)/ Designing the Data Management Component.** The DMC may contain management of and access to persistent data. Examples of classes include ParseVariable, ParseTree, and Parser.
2.2.3.1.2 Five Layers, Five Activities

There are five activities corresponding to the five layers shown in Figure 2.1.13. These activities are: Finding Class-&-Object, Identifying Structures, Identifying Subjects, Defining Attributes, and Defining Services. These five activities can be arranged in any sequence depending on the preferences of an analyst. What is involved in each of these activities is described below.

- **Class-&-Object layer/ Finding Class-&-Objects.** This activity involves finding Classes and Class-&-Objects of the system to be developed. The term ‘Class-&-Object’ means ‘a Class and the Objects in that Class’ (Coad and Yourdon 1991a:53). As each Class or Class-&-Object is found, it is drawn on the OOA/OOD diagram. Figure 2.1.14 shows the notations for Class-&-Object and Class.

![Class-&-Object and Class Notations](Source: Coad and Yourdon 1991a:72)

- **Structure layer/ Identifying Structures.** This activity involves establishing relationships between Class and Class-&-Objects using Generalization-Specialization (Gen-Spec) and Whole-Part structures. Each structure used is added to the OOA/OOD diagram. Figure 2.1.15 shows the notations for the Gen-Spec and Whole-Part structures.
• **Subject layer/ Identifying Subjects.** This activity involves breaking the system into sub-systems. Subjects are used to represent sub-systems. Figure 2.1.16, 2.1.17, and 2.1.18 show the three different types of notations that can be used for the subjects.

1. Subject1

2. Subject2

*Figure 2.1.16: Subject notations, collapsed (Source: Coad and Yourdon 1991a:112)*
Figure 2.1.17: Subject notations, partially expanded (a CASE tool option)
Source: Coad and Yourdon 1991a:112)

Figure 2.1.18: Subject notations, expanded (when shown with other layers)
Source: Coad and Yourdon 1991a:112)

- Attributes layer/ Defining Attributes. This activity involves identifying the attributes for Classes and Class-&-Objects in the system and establishing Instance Connections between objects. Attributes found and Instance Connections established are added to the OOA/OOD diagram. Figure 2.1.19 shows the notations for the attributes and Figure 2.1.20 shows the notations for the Instance Connections.
• **Service layer/ Defining Services.** This activity involves identifying operations for Classes and Class-&-Objects and establishing Message Connections between object. Services found and Message Connections established are added to the OOA/OOD diagram. Figure 2.1.21 shows the notations for the Services and Figure 2.1.22 shows the notations for the Message Connections.

![Diagram of Class-&-Object and Class with Attributes]

*Figure 2.1.19: Attributes Notations*
*Source: Coad and Yourdon 1991a:135)*

![Diagram of Instance Connection Notations]

*Figure 2.1.20: Instance Connection Notations*
*Source: Coad and Yourdon 1991a:136)*

![Diagram of Class-&-Object and Class with Services]

*Figure 2.1.21: Service Notations*
*(Source: Coad and Yourdon 1991a:165)*
2.2.3.2 Other Object-Oriented Methods

Summaries of Booch Object-oriented Design (Booch 1991), Object-Oriented Modeling and Design (Rumbaugh et al. 1991), Wirfs-Brock et al. Responsibility-Driven Design (Wirfs-Brock, Wilkerson, and Wiener 1990), Shlaer and Mellor Object-Oriented Systems Analysis (Shlaer and Mellor 1988 and 1992), and Bailin Object-Oriented Requirements Specification Method (Bailin 1989) are given in Appendix B. Several other methods can be found in the following references: Wasserman, Pircher and Muller (1989 and 1990); Jacobson (1987); Robinson (1987); De Champeaux (1991); Johnson and Foote (1988); Lieberherr and Holland (1989); and Lieberherr et al. (1991); Bulman (1991); Ackroyd and Daum (1991); Beck and Cunningham (1989); Cunningham and Beck (1986); Page-Jones, Constantine and Weiss (1990); Wilson (1990); Alabiso (1988); Iivari (1991).
2.3: COMPARISON OF THE TWO TYPES OF METHODOLOGIES

Some authors (Booch 1989; Coad and Yourdon 1991a) see Object-Oriented methodologies as a revolutionary change from structured methodologies while others (Wasserman, Pircher and Muller 1989) see them as an evolutionary process. The question whether Object-Oriented methodologies are different from structured methodologies depends on which structured methodology is used for comparison (Fichman and Kemerer 1992). If the structured methodology used is data-oriented, such as that of Martin (1990), then Object-Oriented methodologies are considered to be similar to structure methodologies. On the other hand, if the structured methodology used for comparison is process-oriented, such as that of DeMarco (1989), then Object-Oriented methodologies are considered to be different from structured methodologies. This section explores the similarities and differences between structured and Object-Oriented methodologies. An object model is compared with a data-oriented model and a process-oriented model. Techniques used are compared. Differences in paradigms are discussed. Technical advantages and disadvantages of the Object-Oriented methodologies, such as useability and speed, are presented. Management view points of Object-Oriented methodologies, such as risks and learning curves, are described. Finally guidelines is given for adopting the Object-Oriented methodologies.

2.3.1 Similarities

The entity model is a data-oriented model. Therefore, the similarities between the structured and Object-Oriented methodologies can be illustrated by comparing the entity model with the object model. These similarities led some people to use the entity model as the starting point for developing an Object model (e.g., Rumbaugh et al. 1991). This section describes these similarities.
2.3.1.1 Entity and Object

Teague and Pidgeon (1985) define *entity (or object)* in the entity model as some real-world thing about which data is recorded. The ISO TC97 report, referred to by Sutcliffe (1991), defines *entity* as anything of interest, concrete or abstract, that includes association among things. From these two definitions, it can be seen that the definition of entity in the entity model is very similar to the definition of object in the object model. The object in the object model is the entity in the entity model with encapsulated operations. In another word, an entity is an object that has no operation (see Figure 2.2.1).

![Diagram of Entity and Object Models]

*Figure 2.2.1: Comparison of the Entity and Object Models*

2.3.1.2 Attributes of Entities and Objects

Teague and Pidgeon (1985) define an attribute in the entity model as a characteristic of an entity (or object) with a name that can take on a value. This is basically the same definition of an attribute (data structure) as in the object model.
2.3.1.3 Entity and Object Types
Howryszkiewycz (1988) defines an entity type (or entity set) as a set of objects having common attributes. From this definition, it can be seen that the object model is an extension of the entity model. In the object model, object type is a set of objects with not only common attributes, but also common operations. For example, the EMPLOYEE entity type may contain attributes such as Name, Date-Of-Birth, and Salary in the entity model. In the object model, object type may contain, in addition to these attributes, operations such as Change-Name, Change-Date-Of-Birth, and Change-Salary (See Figure 2.2.1).

2.3.1.4 Entity and Object Instances
Sutcliffe (1991) defines an entity instance as an actual occurrence of a particular entity type. This definition is also in line with the definition of the object instance in the object model.

2.3.2 Differences
The differences between the structured and Object-Oriented methodologies can be illustrated by comparing the process-oriented with the object models, the techniques used, and the paradigms. In this section these differences are described.

2.3.2.1 Process-oriented versus Object Models
The process-oriented model separates the processes from the data. As the decomposition of processes does not allow encapsulation, processes can directly access a great number of different entities and are not under the control of any one entity. In addition, entities are passive, stored data that are manipulated by active independent processes as shown in Figure 2.2.2.
On the other hand, the object model encapsulates operations within objects. Operations encapsulated within each object can only access the data structures within that object (see Figure 2.2.3). In an Object-Oriented system, object are active, and can therefore, communicate with one another.
2.3.2.2 Top-Down versus Bottom-Up Techniques

Structured methodologies employ top-down techniques while Object-Oriented methodologies apply both top-down and bottom-up techniques. Bottom-up techniques cannot be used within structured methodologies because of a change in a module may require major change to the whole system (see chapter 2). Bottom-up techniques are made possible within Object-Oriented methodologies through encapsulation and information hiding. Encapsulation and information hiding allow a module to be changed without affecting other modules. A bottom-up technique has the potential to improve the reuse of modules.

2.3.2.3 Solution versus Problems (differences in paradigms)

Structured methodologies focus on the solutions to the problems and how to derive the algorithms to solve them. The Object-Oriented methodologies focus on identifying the object required in the problem domain, what functions these objects need to perform and how they should interact with one another. Table 2.2.1 shows the shift of mindset.

*TABLE 2.2.1: SHIFT OF MINDSET
(Adapted from: Henderson-Sellers 1992)*

<table>
<thead>
<tr>
<th>Procedural Mindset</th>
<th>Object-Oriented Mindset</th>
</tr>
</thead>
<tbody>
<tr>
<td>What does a system do?</td>
<td>What objects is the system comprised of?</td>
</tr>
<tr>
<td>What is its purpose?</td>
<td>How can I model the system dynamically using objects, their behavior, and other objects used by these objects?</td>
</tr>
<tr>
<td>How do I achieve this purpose? How do I design and code this system to achieve this functional behavior?</td>
<td>Algorithmic functions deferred.</td>
</tr>
<tr>
<td>Focus on algorithms.</td>
<td></td>
</tr>
</tbody>
</table>

56
2.3.3 Project Management Points of View of Object-Oriented Methodologies

Software in the '90s is becoming larger and more complex due to the high demands from the users. As a result it is harder to develop and maintain systems using conventional methodologies, leading software managers to look for alternatives. The main issues in software development that interest managers are whether software will be completed on time, within budget, and be of high quality and whether maintenance costs will be minimal. Although the technology is still in its infancy, Object Oriented methodologies claim to solve some of these problems. However, if Object-Oriented methodologies are to be accepted by managers, their advantages over conventional methodologies must be proved in industrial-scale applications, and smooth migration paths need to be established.

2.3.3.1 Benefits

There are several benefits of adopting the Object-Oriented technology. These benefits relate to the strengths of this technology and are described below.

2.3.3.1.1 Shorter Development Time

Shorter development time can be achieved through reusability and smooth transformation. By using pre-developed, pre-tested, reusable software components, new complex systems can be built in shorter time compared with designing and writing these components from scratch (Weinberg, Guimaraes and Heath 1990). Smooth transformation eases the development process and thus reduces the development time.
2.3.3.1.2 Higher Quality Software
Higher quality software results from correctness, robustness, reusability, integrity, and increase in modeling power. Correct specifications are important for producing software that conforms to the requirements. Robustness allows software to handle abnormal conditions without crashing. Reusing existing, proven software components eliminates many errors, thereby improving the quality of the system. Integrity protects important data from corruption. Increases in modeling power means that complex systems can be built without losing quality (Weinberg, Guimaraes and Heath 1990).

2.3.3.1.3 Easier Maintenance
Maintenance is made easier through extendibility, and maintainability. Software can easily be extended during maintenance using inheritance. System maintenance is easier because changes can be made within one class without affecting others (Weinberg, Guimaraes and Heath 1990).

2.3.3.1.4 Lower Development and Maintenance Costs
Development and maintenance costs result from extendibility, maintainability, reusability, and smooth transformation. Extendibility allows software to be extended quickly and therefore reducing development and maintenance costs. Maintainability allows programmers to modify a module without affecting others. This reduces human efforts and thus maintenance cost. Reusability allows faster development and changes to software and thus reducing development costs. Smooth transformation allows faster development and thus reducing development cost.
2.3.3.2 Risks

Due to the immaturity of the Object-Oriented technology, some of the risks that managers need to consider when adopting this approach are:

2.3.3.2.1 Learning Curves

The Object-Oriented way of looking at things is very different from the conventional way. This shift in paradigms may create many problems for traditional programmers who are used to thinking in traditional ways. Traditional programmers may need to unlearn first before learning again and this may take time. The time required for the programmers to be proficient in applying this technology can take up to three months or more (Taylor 1990; Wilkie 1993).

2.3.3.2.2 High Training Cost

The training cost for developers may be high due to the lack of industrial experts and training material (Martin 1993; Weiberg, Guimaraes and Heath 1990; Sommerville 1990). Also, the learning time may be long, which will result in higher costs.

2.3.3.2.3 High Software Cost

Although Object-Oriented designs can be implemented in any conventional language such as C and Pascal, errors are likely to be introduced when mapping from the Object-Oriented design into conventional languages due to the functional-oriented nature of conventional languages (Sommerville 1990). This may defeat the original purpose of the Object-Oriented designs which are supposed to provide improvements over conventional designs. Therefore, new tools may be needed to provide a smooth transition from design to implementation. This may be costly.
2.3.3.2.4 High Design Cost
Object decomposition is difficult. If no experienced Object-Oriented designers are available, the design cost may be higher than it would be for a conventional approach (Sommerville 1990).

2.3.3.2.5 Lack of Supporting CASE Tools
There is insufficient CASE tools currently available on the market to support Object-Oriented methodologies. CASE tools, may be unnecessary for experienced designers, but of great value to inexperienced designers. They may be used to enforce standards for design documentation which help system developers in system maintenance (Sommerville 1990).

2.3.3.2.6 Lack of Tools for Mapping Requirements Specification
In conventional methodologies, functional decomposition is used to aid the production of requirements specifications. Currently, there are no tools for mapping detailed requirements specifications into Object-Oriented design. Full benefits from Object-Oriented methodologies may not be reaped until tools for mapping conventional requirements specification into Object-Oriented design are available on the market (Sommerville 1990).

2.3.4 Guidelines for Adopting Object-Oriented Methodologies
While, there are many potential benefits in Object-Oriented methodologies, equally there are risks that managers need to consider when adopting this technology. Three guidelines are recommended by Booch (1991) in order to adopt Object-Oriented technology successfully:

- Providing a team of developers and managers with the training in the principles of Object-Oriented development.
Applying the Object-Oriented methodologies to a lower risk project first, then allowing the team above to proceed with other projects and to teach other teams the Object-Oriented methodologies.

Giving developers and managers exposure to well-structured Object-Oriented systems. to which could be added

the provision of suitable CASE tools which aid in communication between user analyst and programmer, improve productivity, reduce errors, lower costs and enforce standards.

2.3.5 Conclusion

Structured methodologies, even though they improved the productivity of software systems in the past, have been unable to cope with the size and complexity of the present software systems. This chapter has covered structured methodologies, the reasons for their existence and the problems they face. It also described the history, paradigms, methods, advantages and disadvantages of Object-Orientation, plus the benefits and shortcomings of object-oriented technology. A comparative review of the two methodologies is done as well as the guidelines for applying object-oriented methodologies.

Despite the many risks in adopting the Object-Oriented technology due to the lacks of supporting tools and the high starting costs, the potential of this technology should not be ignored. As this technology is evolving into maturity, many tools will be readily available at cheaper costs. The next chapter will describe the model that amalgamates the properties of both structured and object-oriented methodologies.