Chapter 6

Software Model and Programming Paradigms

6.1 Why Software Model?

Software Model is a necessary thing for successful completion of the software with reduced 'bugs'. It provides a structure which makes easier for problem solving while implementation. It experiments to explore multiple solutions for a single piece of problem. It furnishes abstraction to manage the complexity of the software program. It reduces the delivery time to market thereby overcoming business problems [21]. It mainly reduces the development costs and manages the risk of mistakes. Some software development methods are listed below:-

- Top-Down and Bottom-up Model
- Prototyping Model
- Evolutionary Prototyping Model
- Iterative and Incremental development Model
- Waterfall Model
- Spiral Model
- Chaos Model

Each and every model will be discussed in detail below.

6.1.1 Top – Down and Bottom-up Model:

Top-down and Bottom-up are approaches to the software development process, and by extension to other procedures, mostly involving software. In the Top-Down Model an overview of the system is formulated, without going into detail for any part of it. Each part of the system is then refined by designing it in more detail. Each new part may then be refined again, defining it in yet more detail until the entire specification is detailed enough to begin development [21].

6.1.2 Prototyping Model:

The prototyping model is a software development process that begins with requirements collection, followed by prototyping and user evaluation. Often the end users may not be able to provide a complete set of application objectives, detailed input, processing or output requirements in the initial state. After the user evaluation, another prototype will be built based on feedback from users, and again the cycle returns to customer evaluation [21].

6.1.3 Evolutionary Prototyping Model:

Code and fix development is not so much a deliberate strategy as an artefact of schedule pressure on software developers. Without much in the way of a design, programmers immediately begin producing code. At some point, testing begins (often late in the development cycle), and the inevitable bugs must then be fixed before the product can be shipped [21].

6.1.4 Iterative and Incremental development:

Iterative and Incremental development is a software development process, one of the practices used in Extreme Programming. The basic idea behind iterative enhancement is to develop a software system incrementally, allowing the developer to take advantage of what was being learned during the development of earlier, incremental, deliverable versions of the system. Learning comes from both the development and use of the system, wherever possible. Key steps in the process were to start with a simple implementation of a subset of the software requirements and iteratively enhance the evolving sequence of versions until the full system is implemented. In each iteration, design modifications are made along with addition of new functional capabilities [21].

6.1.5 Waterfall Model:

The Waterfall model is a software development model first proposed in 1970 by W.W.Royce, in which development is seen as flowing steadily through the phases of *requirements analysis, design, implementation, testing (validation), integration, and maintenance* [21].

In the original article, Royce advocated using the model repeatedly, in an iterative way. However, most people do not know that some have discredited for use as a Real World process. In practice, the process rarely proceeds in a purely linear fashion. Iterations, by going back to or adapting results of the precedent state, are common.

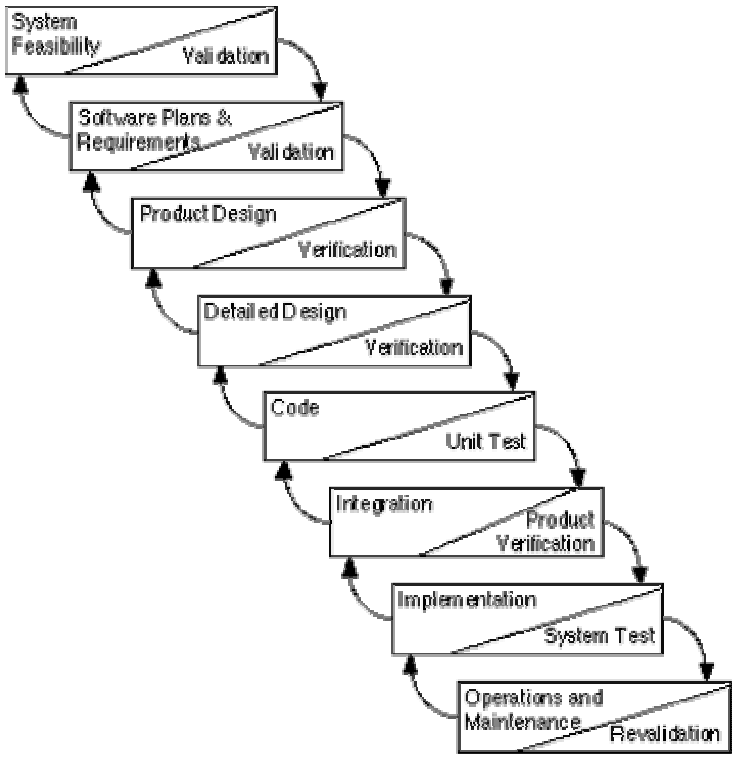

Figure 19: Waterfall Model

The below table shows the explanation about each stages in water fall model.

Feasibility:	Defining a preferred concept for the software product, and determining its life-cycle feasibility and superiority to alternative concepts.
Requirements:	A complete, verified specification of the required functions, interfaces, and performance for the software product.

Product Design:	A complete verified specification of the overall hardware-software architecture, control structure, and data structure for the product, along with such other necessary components as draft user's manuals and test plans.
Detailed Design:	A complete verified specification of the control structure, data structure, interface relations, sizing, key algorithms, and assumptions of each program component.
Coding:	A complete, verified set of program components.
Integration:	A properly function software product composed of the software components.
Implementation:	A fully functioning operational hardware-software system, including such objectives as program and data conversion, installation, and training.
Maintenance:	A fully functioning update of the hardware-software system repeated for each update.
Phaseout:	A clean transition of the functions performed by the product to its successors.

6.1.6 Spiral Model:

The spiral model is a software development model combining elements of both design and prototyping-in-stages, so it's a healthy mix of top-down and bottom-up concepts. The spiral model was defined by Barry Boehm. This model was not the first model to discuss iteration, but it was the first model to explain why the iteration matters. As originally envisioned, the iterations were typically 6 months to 2 years long. This persisted until around 2000.

However, this model "does not work well for many classes of software, particularly interactive end-user applications." Specifying the requirements for such applications is notoriously difficult since interface design is highly subjective and clients rarely ever understand the real needs the software should meet. As a result, "document-driven standards have pushed many projects to write elaborate specifications of poorly understood user interfaces and decision-support functions, followed by the design and development of large quantities of unusable code". The problem is that a contract is signed before the real requirements of the system are properly understood [22].

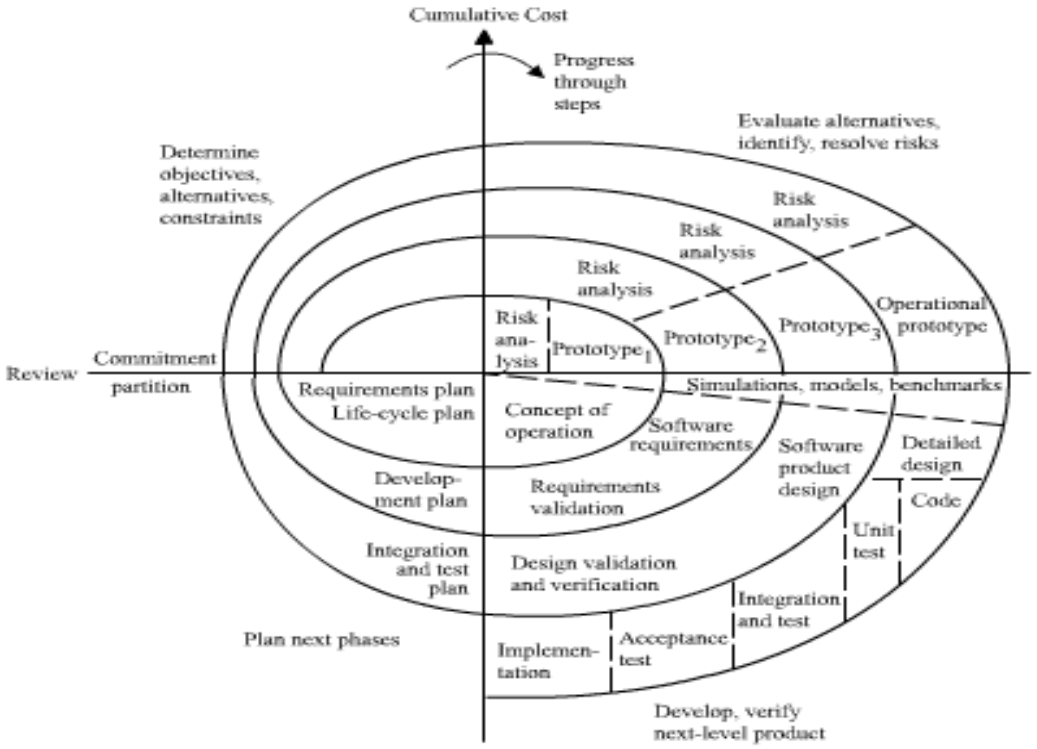

Figure 20 : Spiral Model

6.1.7 Chaos Model:

In computing, the Chaos model is a structure of software development that extends the spiral model and waterfall model. The chaos model was defined by L.B.S. Raccoon. The chaos model notes that the phases of the lifecycle apply to all levels of projects, from the whole project to individual lines of code [21].

- The whole project must be defined, implemented and integrated.
- Systems must be defined, implemented and integrated.
- Modules must be defined, implemented and integrated.
- Functions must be defined, implemented and integrated.
- Lines of code are defined, implemented and integrated.

6.2 Aspects of programming paradigms

The term *"programming paradigm"* is extremely fuzzy because it is often used to capture a set of different software-related aspects under a particular catch-phrase. These different aspects are often located on different levels of abstraction and their interrelationships are seldom explicitly formulated. Here a triangle is shown in the following figure to describe the different levels of abstraction that is the view made up a programming paradigm. The form of a triangle express the fact that the number of concepts on a particular level of abstraction increases on higher levels. Furthermore, a layered approach is quite common in computer science theories to clearly separate the concepts on different levels of abstraction.

The main advantage of a layered approach is that no knowledge of lower levels is necessary to understand and to work with higher level concepts because ideally, each level of abstraction represents a conceptually closed framework. In reality, unfortunately, the higher level theories are not only much more complex then lower level ones, but they are often incomplete. Therefore, it often becomes necessary to combine several higher level theories to obtain a full coverage of the intended part of the world that should be modelled.

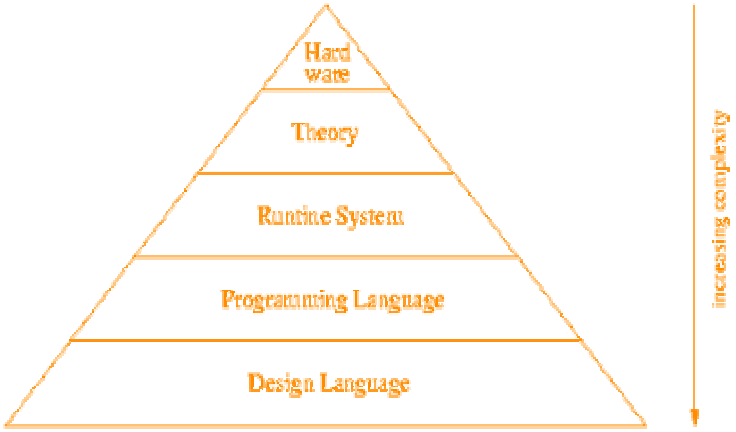

Figure 21 : Aspects of Programming Paradigm

Furthermore, that the distinctions between the different levels are not too sharp. Because, *"Most programming models are assumed to be essentially equal in their computational power" (Church's thesis),* any programming model can be implemented in terms of any other mode [21]l.

Thus, it is possible to write object-oriented software in a purely imperative programming language or to implement a deductive database in an object-oriented framework. In the following sections, a break-down of concepts are shown that clearly separates intra-model aspects and that allows for an inter-model comparison of these concepts. Some concepts can be shifted along the abstraction hierarchy, but the current assignment to a particular level is adequate.

6.2.1 Hardware

The first level of abstraction encapsulates the architecture that is implemented in the computer hardware. Today, most computers still have the von Neumann architecture that was introduced in the late 1940s. The architecture consists of a *processor* that is subdivided into units for computation and control and a *memory* store that holds the instructions and the data of the program [21].

This architecture is still common in modern computers although it has been greatly optimized by using techniques such as pipelining, caching or parallelism to speed up computation. A recent trend in the hardware community is to turn away from integrated, large-scale systems and towards networks of normal personal computers that jointly work on a computationally demanding task. These virtual supercomputers combine the advantage of lower costs through the use of standard hardware with an extreme scalability that allows to add more computational resources whenever this is necessary.

6.2.2 Theories

On the next higher level of abstraction, however, things are different. Theories are conceptualizations of a particular computational model that abstracts away from the characteristics of the hardware. The first theories were aimed at capturing the in-principle capability of a computational device in order to allow for general statements about what can be automatically computed and what cannot. Turing's theory, for example, is a radical mathematical conceptualization of the von Neuman architecture that enables us to formally analyze all possible programs that can be executed on such architecture. Other computational theories are intended as tools to help the programmer to express the ideas of what a program is supposed to do more naturally.

6.2.3 Runtime System

The runtime system of a particular programming paradigm provides the environment for the program interpretation and these environments can be radically different. In the more simple forms, they are restricted to administrative tasks such as managing the heap or they provide slightly more elaborate services such as garbage collection.

However, there also exist very complex runtime environment that provide complete reasoning engines for logic programming that are for example used in declarative programming languages such as Prolog. Objects and agents and the various relationships that exist between them within their respective programming model are conceptual abstractions that require an implementation such that they can be used by higher levels of abstraction [21].

6.2.4 Programming Language

In this level of abstraction, the syntactical framework for the manipulation of the entities on the runtime level is defined. The programs that are written in a particular programming language are either directly interpreted by the runtime system or they are compiled into an intermediate format that is understood by the runtime system or directly to assembler code.

The syntactical constructs that are provided by the programming language should allow the programmer to use the underlying semantic concepts efficiently and to express the intended functionality of the program elegantly. For example, it is generally possible to implement a particular conceptual model with any general purpose language, e.g. it is possible to write object-oriented programs in C, but in general, it is much easier and more comfortable for the programmer if the terms of the conceptual framework can be used directly. Even an integration of several conceptual models into a single high level programming language can be problematic as is often difficult to find a good combination of concepts that is not overwhelming for the average user and then to find a concise syntactical representation for these different concepts [21].

6.2.5 Design Language

Design languages are further abstractions from a particular programming language that aim at the conceptual modelling of a system at a more coarse grained level. Design languages often use graphical notations that make it easier from the designer to access the overall system structure. Probably the currently best known design language is the *Unified Modelling Language (UML)* that tries to integrate several, until then separated design notations under a common hat. Due to the general nature of the core UML, it is not always suited for all problem areas, and therefore, extensions that cover special aspect have already been proposed [21 & 22].

In a more general sense, however, design languages should not necessarily be constraint to modelling aspects of the system. The reason for this view is that these frameworks provide their own set of structural abstractions that represent a "language" on this particular level of abstraction.

6.3 Types of programming paradigms

Various available programming paradigms are listed below

- Procedural Programming
- Structured Programming
- Imperative Programming
- Declarative Programming
- Functional Programming
- Literate Programming
- Object Oriented Programming
- Concurrent Programming
- Component Oriented Programming

Each sections are discussed below [21].

6.3.1 Procedural Programming:

Procedural Programming is a method of computer programming based upon the concept of the unit and scope. A procedural program is composed of one or more units or modules – either user coded or provided in a code library. Each module is composed of one or more procedures, also called a function, routine, subroutine, or method depending on programming language. It is possible for a procedural program to have multiple levels or scopes, with procedures defined inside other procedures. Each scope can contain variables which cannot be seen in outer scopes.

6.3.2 Structured Programming:

Structured Programming can be seen as a subset or sub discipline of procedural programming, which is one of the major programming paradigms. It is most famous for removing or reducing reliance on the GOTO statement.

Historically, several different structuring techniques or methodologies have been developed for writing structured programs. The two most common are Jackson Structured Programming which is based on aligning data structures with program structures and Dijkstra's structured programming, which is based on splitting programs in to sub-sections, each with a single point of entry and of exit.

6.3.3 Imperative programming:

In computer science, imperative programming, as opposed to declarative programming, is a programming paradigm that describes computation in terms of a program state and statements that change the program state. In much the same way as the imperative mood in natural languages expresses commands to take action; imperative programs are a sequence of commands for the computer to perform. The hardware implementation of almost all computers is imperative; nearly all computer hardware is designed to execute machine code, which is native to the computer, written in the imperative style. From this low-level perspective, the program state is defined by the contents of memory, and the statements are instructions in the native machine language of the computer.

6.3.4 Declarative programming:

Declarative Programming is an approach to computer programming that takes a different approach from traditional imperative programming in Fortran, C++ or Java. Whereas imperative programming gives the computer a list of instructions to execute in a particular order, declarative programming describes to the computer a set of conditions and lets the computer figure out how to satisfy them. Declarative programming includes both functional programming and logic programming.

6.3.5 Functional programming:

Functional programming is a programming paradigm that treats computation as the evaluation of mathematical functions. In contrast to imperative programming, functional programming emphasizes the evaluation of functional expressions, rather than execution of commands. The expressions in these languages are formed by using functions to combine basic values.

6.3.6 Literate programming:

Literate programming is a certain way of writing computer programs. It is seen as communications to human beings, as works of literature, hence the name "literate programming".

6.3.7 Object oriented programming:

Object –oriented programming (OOP) is a computer programming paradigm that emphasizes the following aspects:

- Objects – packaging data and functionality together into units within a running computer program; objects are the basis of modularity and structure in an object – oriented computer program.
- Abstraction – The ability for a program to ignore some aspects of the information it's manipulating, i.e the ability to focus on the essential. Each object in the system serves as a model of an abstract "actor" that can perform work, report on and change its state, and

"communicate" with other objects in the system, without revealing how these features are implemented. Processes, functions or methods may also be so abstracted, and when they are, a variety of techniques are required to extend an abstraction:

- Encapsulation – Also called information hiding: Ensures that objects cannot change the internal state of other objects in unexpected ways; only the object's own internal methods are allowed to access its state. Each type of object exposes an interface to other objects that specifies how other objects may interact with it. This prevents users from breaking the invariants of the program.
- Polymorphism – References to and collections of objects may refer to objects of different types, and invoking an operation on a reference will produce behaviour depending on the actual type of the referent.
- Inheritance – Organizes and facilitates polymorphism and encapsulation by permitting objects to be defined and created that are specialized types of already-existing objects- these can share (and extend) their behaviour without having to re implement that behaviour. This is typically done by grouping objects into classes, and defining classes as extensions of existing classes, thus and grouping classes into trees or lattices reflecting behavioural commonality.

It is in fact a set of ideas which mostly existed before. They have been brought together, with associated terminology, to create a programming framework. Together the ideas behind OO are said to be so powerful they create a Paradigm shift in programming. The exact definitions of these have some variation depending on point of view.

6.3.8 Concurrent programming:

Parallel programming (also concurrent programming), is a computer programming technique that provides for the execution of operations concurrently, either within a single computer, or across a number of systems. In the latter case, the term distributed computing is used. Multiprocessor machines achieve better performance by taking advantage of this kind of programming. In parallel programming, single tasks are split into a number of subtasks that can be computed relatively independently and then aggregated to form a single coherent solution. Parallel programming is most effective for tasks that can easily broken down into independent tasks such as purely mathematical problems, e.g. factorisation.

6.3.9 Component-oriented programming:

Software componentry is a field of study within software engineering. It builds on prior theories of software objects, software architectures, software frameworks and software design patterns, and the extensive theory of object-oriented programming and object-oriented design of all these. It claims that software components, like the idea of a hardware component used e.g. in telecommunication, can be ultimately made interchangeable and reliable. Software componentry is a common and convenient means for inter-process communication (IPC).

6.4 Project Requirement

Our project "Design and Implementation of a media entertainment system running on a Carputer" requires *'Waterfall Model'* with an *'Object oriented programming'* approach should be done in Delphi 7.0 as specified in the communicated requirements.