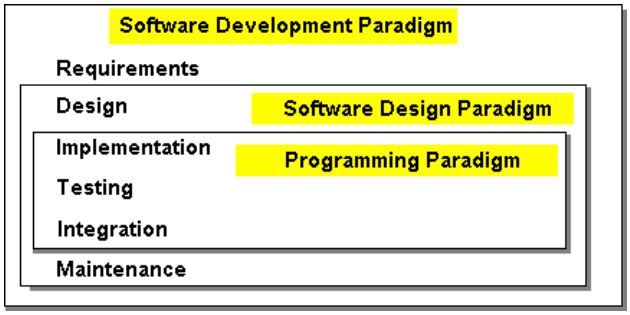
2 SOFTWARE PARADIGMS

Paradigm

"Paradigm" (a Greek word meaning example) is commonly used to refer to a category of entities that share a common characteristic.
We can distinguish between three different kinds of Software Paradigms:

- Programming Paradigm is a model of how programmers communicate a calculation to computers
- Software Design Paradigm is a model for implementing a group of applications sharing common properties
- Software Development Paradigm is often referred to as Software Engineering, may be seen as a management model for implementing big software projects using engineering principles.



Programming Paradigm

A Programming Paradigm is a model for a class of Programming Languages that share a set of common characteristics.
A programming language is a system of signs used to communicate a task/algorithm to a computer, causing the task to be performed. The task to be performed is called a computation, which follows precise and unambiguous rules.
At the heart of it all is a fundamental question: What does it mean to understand a programming language? What do we need to know to program in a language? There are three crucial components to any language.
The language paradigm is a general principle that is used by a programmer to communicate a task/algorithm to a computer.
The syntax of the language is a way of specifying what is legal in the phrase structure of the language; knowing the syntax is analogous to knowing how to spell and form sentences in a natural language like English. However, this doesn't tell us anything about what the sentences mean.
The third component is the semantics, or meaning, of a program in that language. Ultimately, without semantics, a programming language is just a collection of meaningless phrases; hence, semantics is a crucial part of a language.
There have been a large number of programming languages. Back in the '60s, there were over 700 of the – most were academic, special purpose, or developed by an organization for their own needs.

Fortunately, there are just four major programming language paradigms:

- Imperative (Procedural) Paradigm (Fortran, C, Ada, etc.)
- Object-Oriented Paradigm (Smalltalk, Java, C++)
- Logic Paradigm (Prolog)
- Functional Paradigm (Lisp, ML, Haskell)

Generally, a selected Programming Paradigm defines the main property of software developed utilizing a programming language supporting the paradigm.

- scalability/modifiability
- integrability/reusability
- portability
- performance
- reliability
- ease of creation

Software Design Paradigm

Software Design Paradigm embodies the results of people's ideas on how to construct programs, combine them into large software systems, and formal mechanisms for how those ideas should be expressed.
Thus, we can say that a Software Design Paradigm is a model for a class of problems that share a set of common characteristics.

Software design paradigms can be sub-divided as:

- Design Patterns
- Components
- Software Architecture
- Frameworks

It should be especially noted that a particular Programming Paradigm essentially defines software design paradigms. For example, we can speak about Object-Oriented design patterns, procedural components (modules), functional software architecture, etc.

Design Patterns
A design pattern is a proven solution for a general design problem. It consists of communicating 'objects' that are customized to solve the problem in a particular context.
Patterns have their origin in object-oriented programming where they began as collections of objects organized to solve a problem. There isn't any fundamental relationship between patterns and objects; it just happens they began there. Patterns may have arisen because objects seem so elemental, but the problems we were trying to solve with them were so complex.

- Architectural Patterns: An architectural pattern expresses a fundamental structural organization or schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them.
- Design Patterns: A design pattern provides a scheme for refining the subsystems or components of a software system, or the relationships between them. It describes commonly within a particular context.
- Idioms: An idiom is a low-level pattern specific to a programming language. An idiom describes how to implement particular aspects of components or the relationships between them using the features of the given language.

Components

Often equated to design patterns with Emphasis on reusability
Software components are binary units of independent production, acquisition, and deployment that interact to form a functioning program.
A component is a physical and replaceable part of a system that conforms to and provides the realization of a set of interfaces...typically represents the physical packaging of otherwise logical elements, such as classes, interfaces, and collaborations
A component must be compatible and interoperate with a whole range of other components.
Examples of components: “Window”, “Push Button”, “Text Editor”, etc.
Two main issues arise concerning interoperability information:

1. how to express interoperability information (e.g. how to add a “push button” to a “window”;
2. how to publish this information (e.g. library with API reusable via an “include” statement)?

Software Architecture

Software architecture is the structure of the components of the solution. A particular software architecture decomposes a problem into smaller pieces and attempts to find a solution (Component) for each piece. We can also say that architecture defines a software system component, its integration, and interoperability:

- Integration means the pieces fit together well.
- Interoperation means that they work together effectively to produce an answer.

There are many software architectures. Choosing the right one can be a difficult problem in itself.

Frameworks

A software framework is a reusable mini-architecture that provides the generic structure and behavior for a family of software abstractions, along with a context of metaphors that specifies their collaboration and use within a given domain.
Frameworks can be seen as an intermediate level between components and software architecture.
Example: Suppose an architecture of a WBT system reuse such components as “Text Editing Input object” and “Pushbuttons”. A software framework may define an “HTML Editor” which can be further reused for building the architecture.

Software Engineering and Software Paradigms

The term "software engineering" was coined in about 1969 to mean "the establishment and use of sound engineering principles to economically obtain software that is reliable and works efficiently on real machines".
This view opposed the uniqueness and "magic" of programming to move the development of software from "magic" (which only a select few can do) to "art" (which the talented can do) to "science" (which supposedly anyone can do!).
There have been numerous definitions given for software engineering (including those above and below).
Software Engineering is not a discipline; it is an aspiration, as yet unachieved. Many approaches have been proposed including reusable components, formal methods, structured methods, and architectural studies. These approaches chiefly emphasize the engineering product; the solution rather than the problem it solves.
Software Development current situation:

- People developing systems were consistently wrong in their estimates of time, effort, and costs
- Reliability and maintainability were difficult to achieve
- Delivered systems frequently did not work
    - 1979 study of a small number of government projects showed that:

    - 2% worked
    - 3% could work after some corrections
    - 45% delivered but never successfully used
    - 20% used but extensively reworked or abandoned
    - 30% paid and undelivered
- Fixing bugs in delivered software produced more bugs
- Increase in size of software systems
    - NASA
    - StarWars Defense Initiative
    - Social Security Administration
    - financial transaction systems
- Changes in the ratio of hardware to software costs
    - early 60's - 80% hardware costs
    - middle 60's - 40-50% software costs
    - today - less than 20% hardware costs
- The increasingly important role of maintenance
    - Fixing errors, modification, adding options
    - Cost is often twice that of developing the software
- Advances in hardware (lower costs)
- Advances in software techniques (e.g., users interaction)
- Increased demands for software
    - Medicine, Manufacturing, Entertainment, Publishing
- Demand for larger and more complex software systems
    - Airplanes (crashes), NASA (aborted space shuttle launches),
    - "ghost" trains, runaway missiles,
    - ATMs (have you had your card "swallowed"?), life-support systems, car systems, etc.
    - US National security and day-to-day operations are highly dependent on computerized systems.

Manufacturing software can be characterized by a series of steps ranging from concept exploration to final retirement; this series of steps is generally referred to as a software lifecycle.
Steps or phases in a software lifecycle fall generally into these categories:

- Requirements (Relative Cost 2%)
- Specification (analysis) (Relative Cost 5%)
- Design (Relative Cost 6%)
- Implementation (Relative Cost 5%)
- Testing (Relative Cost 7%)
- Integration (Relative Cost 8%)
- Maintenance (Relative Cost 67%)
- Retirement

Software engineering employs a variety of methods, tools, and paradigms.
Paradigms refer to particular approaches or philosophies for designing, building, and maintaining software. Different paradigms each have their advantages and disadvantages which make one more appropriate in a given situation than perhaps another (!).
A method (also referred to as a technique) is heavily dependent on a selected paradigm and may be seen as a procedure for producing some result. Methods generally involve some formal notations and processes.
Tools are automated systems implementing a particular method.

Thus, the following phases are heavily affected by selected software paradigms

- Design
- Implementation
- Integration
- Maintenance

The software development cycle involves the activities in the production of a software system. Generally, the software development cycle can be divided into the following phases:

- Requirements analysis and specification
- Design
    - Preliminary design
    - Detailed design
- Implementation
    - Component Implementation
    - Component Integration
    - System Documenting
- Testing
    - Unit testing
    - Integration testing
    - System testing
- Installation and Acceptance Testing
- Maintenance
    - Bug Reporting and Fixing
    - Change requirements and software upgrading
- Software lifecycles that will be briefly reviewed include:
    - Build and Fix model
    - Waterfall and Modified Waterfall models
    - Rapid Prototyping
    - Boehm's spiral model

Build and Fix model

This works OK for small, simple systems, but is completely unsatisfactory for software systems of any size. It has been shown empirically that the cost of changing a software product is relatively small if the change is made at the requirements or design phases but grows large at later phases.
The cost of this process model is far greater than the cost of a properly specified and designed project. Maintenance can also be problematic in a software system developed under this scenario.

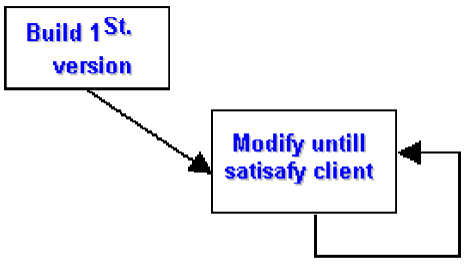


Figure: Build and Fix model

Waterfall and Modified Waterfall models

Waterfall Model

Derived from other engineering processes in 1970. Offered a means of making the development process more structured. Expresses the interaction between subsequent phases.

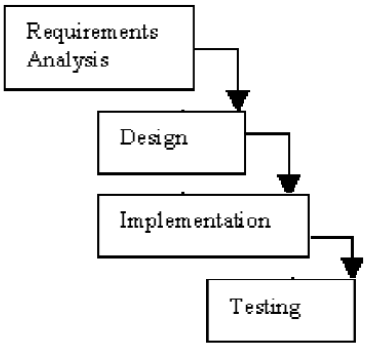


Figure: Waterfall model

Each phase cascades into the next phase. In the original waterfall model, a strict sequentially was at least implied. This meant that one phase had to be completed before the next phase was begun.

It also did not provide feedback between phases or for updating/re-definition of earlier phases. Implies that there are definite breaks between phases, i.e., that each phase has a strict, non-overlapping start and finish and is carried out sequentially.

Critical point is that no phase is complete until the documentation and/or other products associated with that phase are completed.

2.2.2 Modified Waterfall Model

Needed to provide for overlap and feedback between phases. Rather than being a simple linear model, it needed to be an iterative model. To facilitate the completion of the goals, milestones, and tasks, it is normal to freeze parts of the development after a certain point in the iteration. Verification and validation are added. Verification checks that the system is correct (building the system right). Validation checks that the system meets the users' desires (building the right system).

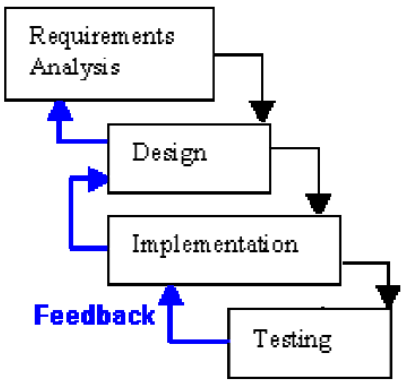


Figure: Modified Waterfall model

The waterfall model (and modified waterfall model) are inflexible in the partitioning of the project into distinct phases. However, they generally reflect engineering practice.

Considerable emphasis must be placed on discerning users' needs and requirements before the system is built. The identification of users' requirements as early as possible, and the agreement between user and developer concerning those requirements, often is the deciding factor in the success or failure of a software project. These requirements are documented in the requirements specification, which is used to verify whether subsequent phases are complying with

the requirements. Unfortunately specifying users' requirements is very much an art, and as such is extremely difficult. Validation feedback can be used to prevent the appearance of a strong divergence between the system under development and the users' expectations for the delivered system. Unfortunately, the waterfall lifecycle (and the modified waterfall lifecycle) are inadequate for realistic validation activities. They are exclusively document-driven models. The resulting design reality is that only 50% of the design effort occurs during the actual design phase with 1/3 of the design effort occurring during the coding activity! This is topped by the fact that over 16% of the design effort occurs after the system is supposed to be completed! In general, the behavior of many individuals in this type of process is opportunistic. The boundaries of phases are indiscriminately crossed with deadlines being somewhat arbitrary.

Rapid Prototyping

Prototyping also referred to as evolutionary development, prototyping aims to enhance the accuracy of the designer's perception of the user's requirements. Prototyping is based on the idea of developing an initial implementation for user feedback, and then refining this prototype through many versions until a satisfactory system emerges. The specification, development, and validation activities are carried out concurrently with rapid feedback across the activities. Generally, prototyping is characterized by the use of very high-level languages, which probably will not be used in the final software implementation but which allow rapid development, and the development of a system with less functionality concerning quality attributes such as robustness, speed, etc.

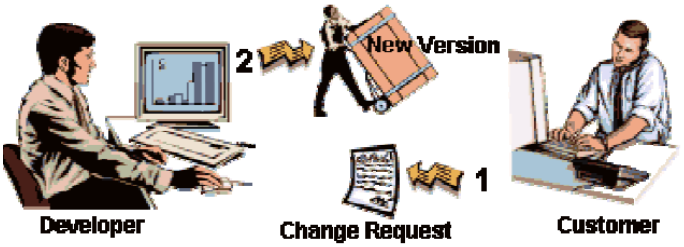


Figure: Rapid Prototyping model

Prototyping allows the clarification of users' requirements through, particularly, the early development of the user interface. The user can then try out the system, albeit a (sub) system of what will be the final product. This allows the user to provide feedback before a large investment has been made in the development of the wrong system.
There are two types of prototypes:

- Exploratory programming: The objective is to work with the user to explore their requirements and deliver a final system. Starts with the parts of the system which are understood, and then evolves as the user proposes new features.
- Throw-away prototyping: The objective is to understand the users' requirements and develop a better requirements definition for the system. Concentrates on poorly understood components.

Experiments with prototyping showed that this approach took 40% less time and resulted in 45% less code; however, it produced code that was not as robust, and therefore more difficult to maintain. Documentation was often sacrificed or done incompletely. The schedule expectations of users and managers tended to be unrealistic, especially for throw-away prototypes.

Boehm's Spiral Model

Need an improved software lifecycle model which can subsume all the generic models discussed so far. Must also satisfy the requirements of management.
Boehm proposed a spiral model where each round of the spiral

- identifies the subproblem which has the highest risk associated with it
- finds a solution for that problem.

Imperative (Procedural) Programming Paradigm

Any imperative program consists of

- Declarative statements give a name to a value. A named value is called a variable. Thus, declarative statements create variables. In procedural languages, it is common for the same variable to keep changing value as the program runs.
- Imperative statements which assign new values to variables
- Program flow control statements which define the order in which imperative statements are evaluated.

Example:

```
var factorial = 1; /*Declarative statement*/
var argument = 5;
var counter = 1;
while (counter <= argument) /*Program flow statement*/
{
factorial = factorial*counter; /*Imperative statement*/
counter++;
}
```

Variables and Types

Different variables in a program may have different types. For example, a language may treat two bytes as a string of characters and as a number. Dividing a string '20' by number '2' may not be possible. A language like this has at least two types - one for strings and one for numbers.

Example:

```
var PersonName = new String(); /*variable type "string"*/
var PersonSalary = new Integer(); /*variable type "integer"*/
```

Types can be weak or strong. Strong type means that at any point in the program when it is running, the type of a particular chunk of data (i.e. variable) is known. Weak type means that imperative operators may change a variable type.

Example:

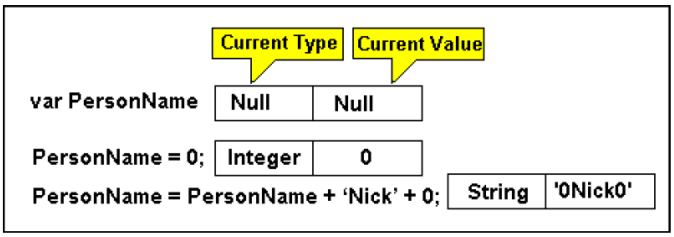
```
var PersonName; /*variable of a weak type"*/
PersonName = 0; /*PersonName is an "integer"*/
PersonName = 'Nick'; /*PersonName is a "string"*/
```

Languages supporting weak variable types need sophisticated rules for type conversions.

Example:

```
var PersonName; /*variable of a weak type"*/
PersonName = 0; /*PersonName is an "integer"*/
PersonName = PersonName + 'Nick' + 0; /*PersonName is a string "0Nick0"*/
```

To support weak typing, values are boxed together with information about their type - value and type are then passed around the program together.



Functions (Procedures)

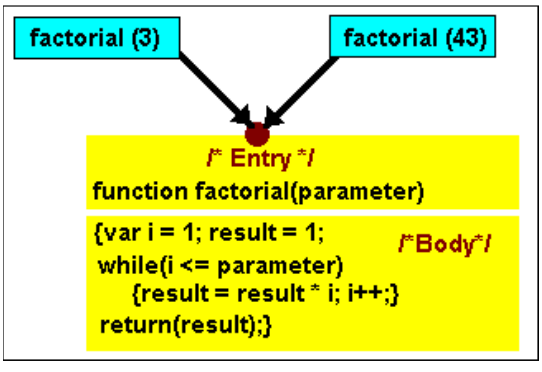
Programmers have dreamed/attempted of building systems from a library of reusable software components bound together with a little new code.

Imperative (Procedural) Programming Paradigm is essentially based on the concept of so-called "Functions" also known as "Modules", "Procedures" or "Subroutines".

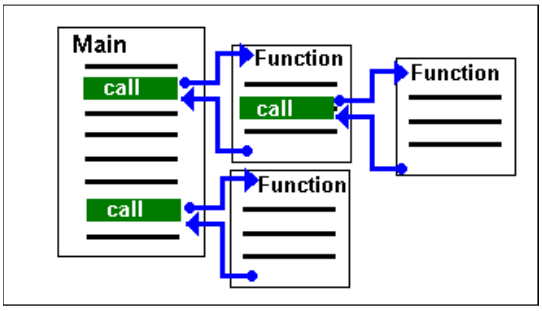
A function is a section of code that is parceled off from the main program and hidden behind an interface:

```
function factorial(parameter)
{
var i = 1;
var result = 1;
while(i <= parameter)
{
result = result * i;
i++;
}
return(result);
}
```

- The code within the function performs a particular activity, here generating a factorial value
- The idea of parceling the code off into a subroutine is to provide a single point of entry. Anyone wanting a new factorial value has only to call the "factorial" function with the appropriate parameters.



Here's what the conventional application based on the Imperative (Procedural) Programming Paradigm looks like:



- The main procedure determines the control flow for the application
- Functions are called to perform certain tasks or specific logic
- The main and sub procedures that comprise the implementation are structured as a hierarchy of tasks.
- The source for the implementation is compiled and linked with any additional executable modules to produce the application

Data Exchange between Functions (Procedures)

When a software system's functionality is decomposed into several functional modules, data exchange/flow becomes a key issue. Imperative (Procedural) Programming Paradigm extends the concept of variables to be used as such data exchange mechanism.

Thus, each procedure may have some special variables called parameters. The parameters are just named place-holders which will be replaced with particular values (or references to existing values) of arguments when the procedure is called.

Example:

```
function main()
{
var argument = 25;
var result = factorial(argument)
/* Note, the imperative operator replaces the “parameter” place holder with a current value of the variable
“argument”*/
}
-----------------------------------------------
function factorial(parameter)
{
var i = 1;
var result = 1;
while(i <= parameter)
{
result = result * i;
i++;
}
return(result);
}
```

Passing arguments to a function

There might be two different techniques for such replacement which are known as passing an argument value and passing an argument reference.

In the case of passing a value, a current argument value is duplicated as a value for a new parameter variable dynamically created for the procedure. In this case, variables used as arguments for calling sub-routines cannot be modified by imperative operators inside of the sub-routines.

In case of passing a reference, the sub-routine gets to control (i.e. reference) to a current value of the argument variable. In this case, variables used as arguments for calling sub-routines can be modified by imperative operators inside of the sub-routines.

Thus, types of variables defined as parameters of a function should be equivalent to (or at least compatible with) types of variables (constants) used as arguments.

Polymorphic Languages

When strong static typing is enforced it can be difficult to write generic algorithms - functions that can act on a range of different types. Polymorphism allows "any" to be included in the type system. For example, the types of a list of items are unimportant if we only want to know the length of the list, so a function can have a type that indicates that it takes lists of "any" type and returns an integer. Moreover, polymorphism allows combining functions implemented employing different programming languages supporting potentially different types of variables.

Pragmatically speaking, polymorphic languages allow defining new types as hidden functions which should be automatically applied to values of such “user-defined type” to convert it to values of a “standard” language type.

Variable Scope

Normally, variables that are defined within a function, are created each time the function is used and destroyed again when the function ends. The value that the function returns is not destroyed, but it is not possible to assign a value to the variable inside the function definition from outside.

Example:

```
function one()
{
var dynamicLocalVariable = 25;
two();
/* at this point just one variable “dynamicLocalVariable” exists */
alert(dynamicLocalVariable);
/* this operator displays the current value “25” */
}
function two()
{
var dynamicLocalVariable = 55;
/* at this point two variables “dynamicLocalVariable” exists */
alert(dynamicLocalVariable);
/* this operator displays the current value “55” */
}
```

Such variables are called dynamic local variables. There may be also so-called static local variables. Static local variables that are defined within a function, are created only once when the function is used for the first time. The value of such variable is not destroyed and can be reused when the function is called again.

Example:

```
function one()
{
var x = two();
alert(x);
/* this operator displays the current value “10” */
x = two();
alert(x);
/* this operator displays the current value “20” */
}
function two()
{
var static staticLocalVariable = 0;
staticLocalVariable = staticLocalVariable + 10;
return(staticLocalVariable);
}
```

Note that function “two” returns different values for the same set of arguments. Such functions are called reactive functions. Generally, testing and maintenance of projects having many reactive functions become a very difficult task. For practical reasons, many software projects do use some static data.
Note, it is still not possible to assign a value to the local static variable inside a function from outside.
There may be also so-called static global variables. Static global variables that are defined within any function, are created only once when the whole software system is initiated. The value of such a variable is never destroyed and can be reused by imperative operators inside any function.
Example:

```
function one()
{
var global globalLocalVariable = 0;
two();
alert(globalLocalVariable);
/* this operator displays the current value “10” */
two();
alert(globalLocalVariable);
/* this operator displays the current value “20” */
}
function two()
{
globalLocalVariable = globalLocalVariable + 10;
}
```

Here, the function “two” also demonstrates a “reactive” behavior. Maintaining and testing projects heavily based on global variables becomes even more difficult than in the case of local static variables. Nevertheless, for practical reasons many software development paradigms do use such global static variables.

Software Design Methodology (Procedural Paradigm)

Benefits of the Paradigm:

- Re-usability: anyone that needs a particular functionality can use an appropriate module, without having to code the algorithm from scratch.
- Specialization: one person can concentrate on writing the best possible module (function) for a particular task while others look after other areas.
- Upgradability: if a programmer comes up with a better way to implement a module then he/she simply replaces the code within the function. Provided the interface remains the same - in other words, the module name and the order and type of each parameter are unchanged - then no changes should be necessary for the rest of the application.

However procedural modules have serious limitations:

- For a start, there is nothing to stop another programmer from meddling with the code within a module, perhaps to better adapt it to the needs of a particular application.
- There is also nothing to stop the code within the function from making use of global variables, thus negating the benefits of a single interface providing a single point of entry.

The paradigm is best suited for the waterfall model of software development.

Design

A particular software system is viewed in terms of its modules and data flowing between them starting with a high-level view.

In this case, software design methodology can be categorized as a Top-down modular design (functional design viewpoint).

The basic design concepts include:

- Modularity
    - Modules are used to describe a functional decomposition of the system
    - A module is a unit containing:
        - executable statements
        - data structures
        - other modules
    - A module:
        - has a name
        - can be separately compiled
        - can be used in a program or by other modules
    - System design generally determines what goes into a module
- Cohesive
    - Single clearly defined function
    - Description of when and how used
    - Loosely Coupled Modules (Modules implement functionality, but not parts of other modules)
- Black Boxes (information hiding)
    - each module is a black box
    - each module has a set of known inputs and a set of predictable outputs
    - inner workings of the module are unknown to the user
    - can be reusable
- Preliminary and Detailed Design specifies the modules to carry out the functions in the DataFlow Diagrams (DFD).

Preliminary design deals mainly with Structure Charts

Hierarchical tree structure

- Modules - rectangle boxes
- calling relationships are shown with arrows
- arrows are labeled with the data flowing between modules

Module Design

- Title
- Module ID - from structure charts
- Purpose
- Method - algorithm
- Usage - who calls it
- External references - other modules called
- Calling sequence - parameter descriptions
- Input assertion
- Output assertion
- Local variables
- Author(s)
- Remarks

Preliminary Design Document

- Cover Page
- Table of Contents
- Design Description
- Software Structure Charts
- Data Dictionary
- Module Designs
- Module Headers
- Major Data Structures Design
- Design Reviews (Examination of all or part of the software design to find design anomalies)

Overview of Detailed Design

- select an algorithm for each module
- refine the data structures
- produce detailed design document

Implementation

Coding (for each Module)

- Source Code
- Documentation

Integration

- Decide what order the modules will be assembled
- Assemble and test the integration of modules
- After final assembly perform the system test
- Note, coding and testing are often done in parallel

Testing

Types of testing

- Unit testing
- Integration testing
- Acceptance testing

As was mentioned above, the paradigm is best suited for the waterfall model of software development. Implementing change requirements and especially rapid prototyping are weak points of the programming paradigm.