Chapter - 9
Software For Instrumentation And Control Applications

9.1 Types of software, Selection And Purchase

9.2 Software Models and Their Limitations

9.3 Software Reliability

9.4 Fault Tolerance

9.5 Software Bugs and Testing

9.6 Good Programming Practice

9.7 User Interface

9.8 Embedded and Real Time Software

Software For Instrumentation And Control Applications

- Autocad, Labview, Proteous -for drawing electrical diagrams
- Smart Draw -specifically to draw P&IDs.
- Matlab- for signal processing
- Google earth
- Modscan - to test logics -specially for searching bugs.

- Micro-controller Programming suites such as Codevision AVR, Bascom AVR , PIC Basic Pro, arduino IDE & so on.
- PLC Programming Software such as SIEMENS SIMATIC S7, Allen Bradley Studio 5000 or RSLOGIX 5000, XILINX ISE, Vivado for FPGA
- DCS Programming Software such as YOKOGAWA CENTUM, SIEMENS PCS7, EMERSON DeltaV

South Korea Nuclear Plant Operator's Computers

Photo: Embedded Intelligence System, Kepy Cement Pvt. Ltd

9.1 Types of software

- System Software: e.g. operating systems
- Application Software: e.g. focuses field of applications such as MIS
- Real-time control and data processing systems
  - e.g. payroll
- Graphical system
  - e.g. games, CAD

9.1 Selection and Purchase

The **selection of a particular language** depends on

- management directives,
- the knowledge and expertise of the software team,
- hardware and available tools.

- **Purchase the software** after you have defined your software requirements and surveyed vendors for availability, reputation and experience.
- Some qualification of a vendor:
  - Acceptance testing
  - Review of vendor's quality assurance
  - Verification testing
  - Qualification report

- Furthermore, required documentations from a vendor:
  - Requirements specification
  - Interface specification
  - Test plans, procedures, results
  - Configuration management plan
  - Hazard analysis
- *Don't buy cheap software tools just to save money!* You will lose much more money in the long run from wasted time forced by delays and inadequacies of cheap tools.

9.2 Software Models

9.2.1. Waterfall Model

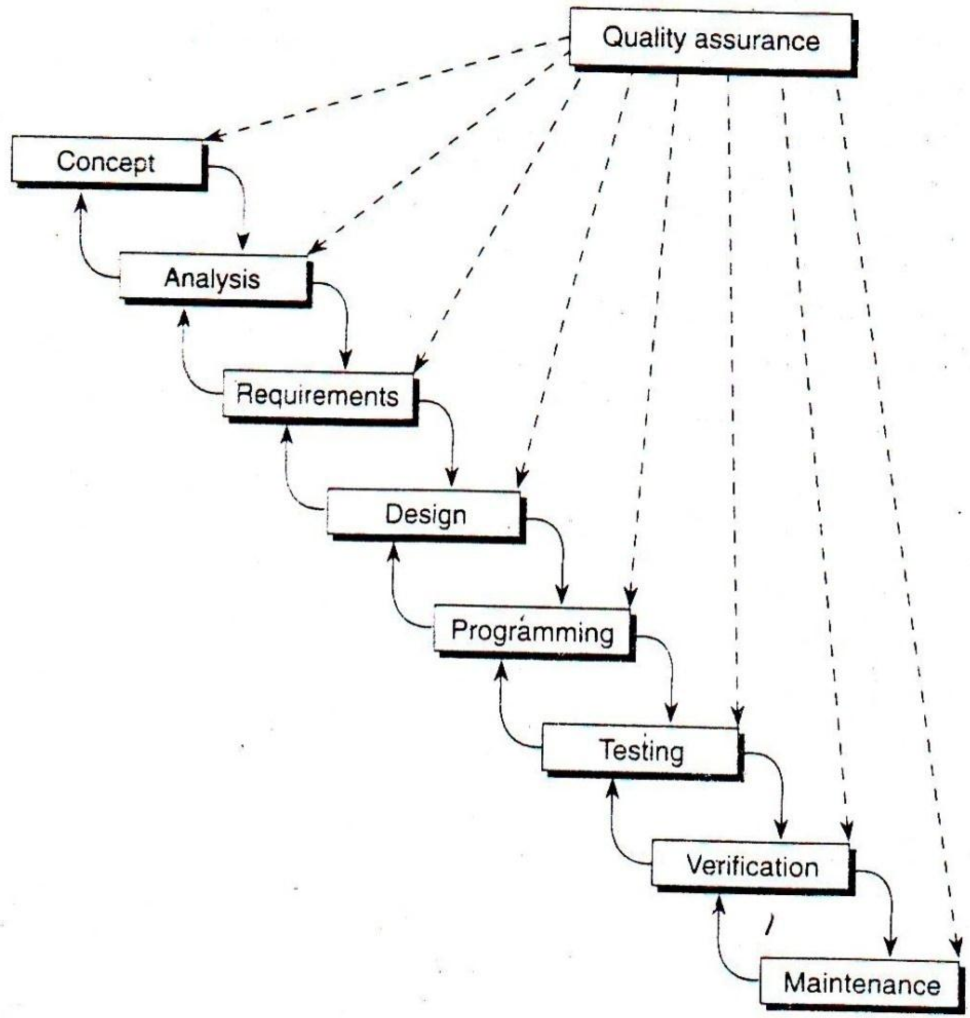


FIG. 11.1 The waterfall model of software development.

Waterfall Model

- The first formally defined software life cycle model was the *waterfall model* [Royce 1970]
- The waterfall model is a software development model in which the results of one activity flowed sequentially into the next as seen as flowing steadily downwards (like a water) through different phases.
- The US Department of Defense contracts prescribed this model for software deliverables for many years, in DOD Standard 2167-A.

Benefits of Waterfall Model

- Managers *love* waterfall models
- Minimizes change, maximizes predictability
- Costs and risks are more predictable
- Highly documented
- Can be used for the projects whose requirements are not changeable
- Each stage has milestones and deliverables: project managers can use to gauge how close project is to completion
- Sets up division of labor: many software shops associate different people with different stages:
  - Systems analyst does analysis,
  - Architect does design,
  - Programmers code,
  - Testers validate, etc.

Limitation of Waterfall Model

- Offers no insight into how each activity transforms artifacts (documents) of one stage into another
  - For example, requirements specification → design documents?
- Fails to treat software a problem-solving process
  - Unlike hardware, software development is not a manufacturing but a creative process
  - Manufacturing processes really can be linear sequences, but creative processes usually involve back-and-forth activities such as revisions
  - Software development involves a lot of communication between various human stakeholders
- Complex documentation requires highly profiled manpower
- Can not be used for changing requirements
- Nevertheless, more complex models often embellish the waterfall,
  - incorporating feedback loops and additional activities

9.2.2 Prototyping Model

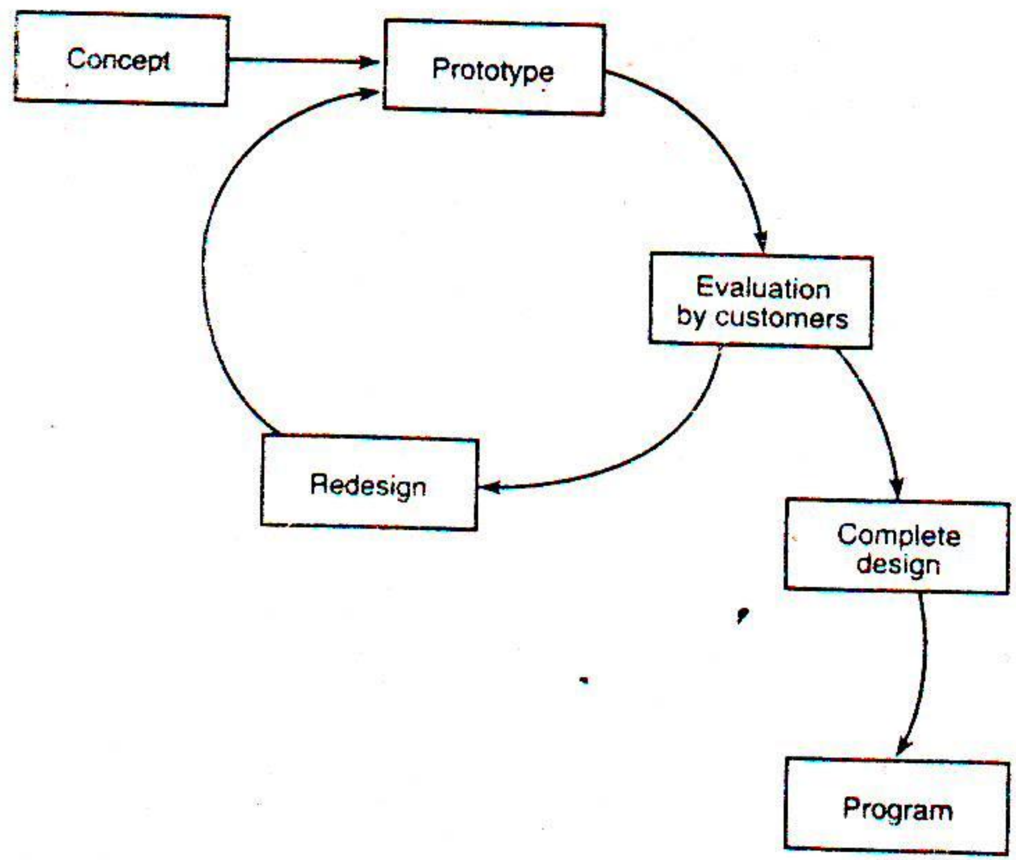


Fig. 11.4 Prototyping model for software development.

Advantages of Prototyping Model

- Due the interaction between the client and developer right from the beginning , the objectives and requirements of the software is well established.
- Suitable for the projects when client has no clear idea about his requirements.
- The client can provide its input during development of the prototype.
- The prototype serves as an aid for the development of the final product.

Limitation of Prototyping Model

- The quality of the software development is compromised in the rush to present a working version of the software to the client.
- Sometimes prototype ends as final product which result in quality + maintenance problem
- Client may divert attention solely to interface issue
- Testing + documentation forgotten
- Designer tends to rush product to market without considering longterm reliability, maintenance, configuration control
- Creeping featurism:

  customer demand change often each evolution

  suitable for small, medium size interactive system

9.2.3.Spiral Model

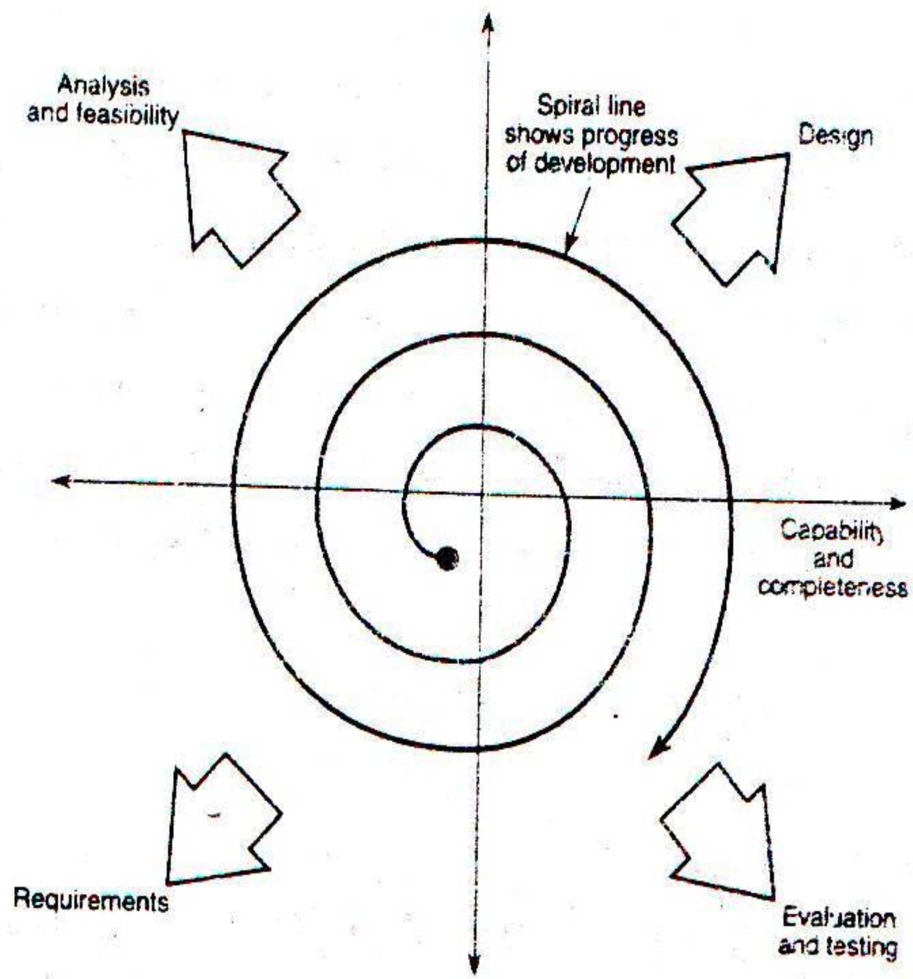


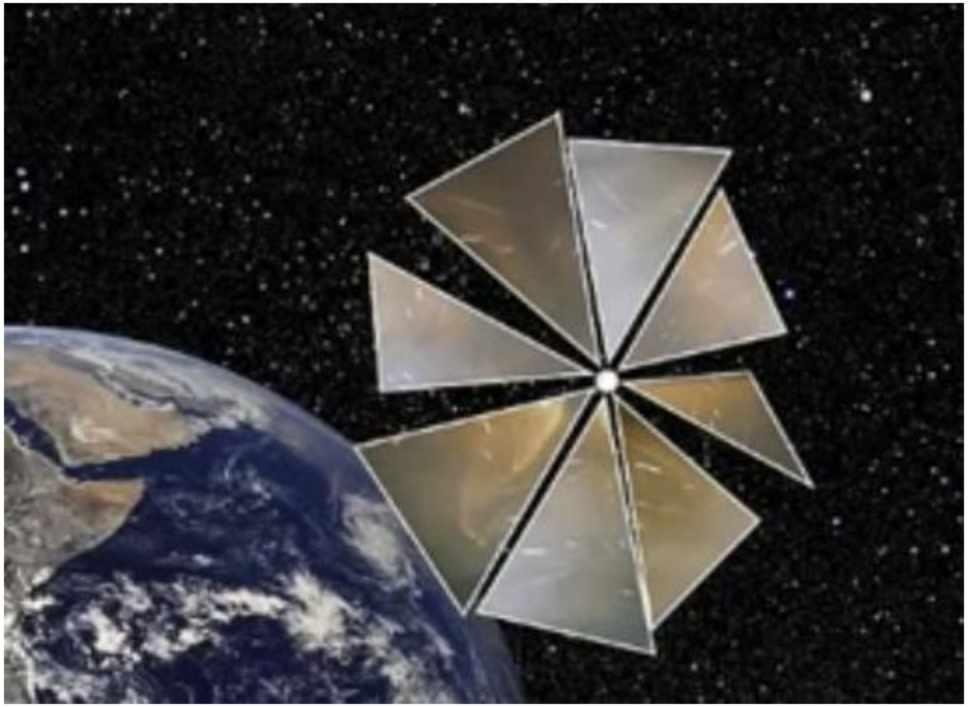
FIG. 11.5 Spiral model of software development.

9.2.3.Spiral Model

- Uses incremental approach
- Combination of waterfall and prototyping model
- Spiral Model – *risk driven rather than document driven*
- Risk is related to the amount and quality of available information. The less information, the higher the risk

- Strengths
  - Introduces risk management
  - Prototyping controls costs
  - Evolutionary development
  - Release builds for beta testing
  - Marketing advantage
- Weaknesses/Limitation
  - Lack of milestones
  - Management is dubious of spiral process
  - Change in Management
  - Prototype Vs Production

9.3. Software Reliability



Failed Satellite Launch, 2005

Airplane Crash, 2002

9.3. Software Reliability

- Software reliability is defined as the probability of failure-free operation of a software system for a specified time in a specified environment.
    - Example
        - The probability that a PC in a store is up and running for eight hours without crash is 0.99.

- Failure intensity is a measure of the reliability of a software system operating in a given environment.
    - Example: An air traffic control system fails once in two years.
- Comparing the two
    - The first puts emphasis on MTTF, whereas the second on count.

Factors Influencing Software Reliability

- A user's perception of the reliability of a software depends upon two categories of information.
  - The number of faults present in the software.
  - The ways users operate the system.
    - This is known as the *operational profile*.
- The fault count in a system is influenced by the following.
  - Size and complexity of code
  - Characteristics of the development process used
  - Education, experience, and training of development personnel
  - Operational environment

9.4 Fault tolerance

- Fault tolerance concerns safety and operational uptime, not reliability.
- It defines how a system prevents or responds to bugs, errors, faults or failures.
- Use
  - Check sums on blocks of memory to detect bit flips
  - Watchdog timer: h/w that monitors a system characteristic to check the control flow and signals the processor with logic pulse when it detect fault.
  - Roll-Back-Recovery or Roll-Forward-Recovery
  - Careful design
  - Redundant architecture

9.5 Software Bugs and Testing

9.5.1. PHASES OF BUGS

1.INTENT:

- Wrong assumption or misunderstanding
- Correctly solving wrong problems
- Viruses
- Slang-limits of operation too broadly or too narrowly defined

2. TRANSLATION

- Incorrect algorithm
- Incorrect analysis
- Misinterpretation

3. **EXECUTION**

- Semantic error –does not know how command works
- Syntax error- rules of language
- Logic error- using wrong decision
- Range error-overflow /underflow error
- Truncation error- incorrect rounding
- Data error –not initialing values, wrong error etc
- Language misuse-inefficient coding
- Documentation-wrong/misleading comments

4. OPERATION

- Changing paradigm
- Interface error
- Performance
- Hardware failure
- Human error

9.5.2 Software Testing:

9.5.2 Software Testing:

A) Internal reviews

- By colleagues examine the correctness of S/w and can figure out mistakes and error in logic
- More than 50% of errors, can be found and correct by code inspection or audit

B) Black box testing:

I/P- O/P interface, are functioning correctly without concerning what happens in S/w.

C) White box testing:

-Exercises all logical decisions and functional path within S/w module

-requires intimate knowledge of S/w module

-exhaustive testing is impossible

may take 100 of years to test each and every possible combination

D) **Alpha and Beta testing:**

- Type of black box testing in actual environment
- **Alpha testing-** programmer collaborate with user
- **Beta testing**- user isolated from programmer
- Verification- debuggers, logic analyzer, in circuit analyzer in circuit debugger etc.

9.6. Good Programming Practice

- For useful, reliable, maintainable program we must make them readable and understandable.

A) Style and format

- program- to do something

  - to communicate designer’s intent to other structure of program and comment.

B) Design:

- Documentation from beginning
- Pseudo code before program
- Keep routine short
- Write clearly: don’t sacrifice clarity for efficiency
- Make routine right, clear, simple and correct before making it faster

C) Comments:

- Readable and clear
- Should not be paraphrase of code
- Write a prologue for each routine
- Should be correct (incorrect comments are worse than no comment)
- Comment more than you think you need

D) Variables:

- Name properly
- Minimize use of global variables
- Don’t pass pointer
- Pass intact values

Structured Programming:

- Establish framework for generating code that is more readable, useful, reliable and maintainable.
- Isolate device dependent code for simplicity and reuse.
- Large modules: divide among team for more productive and parallel effort.
- Use of library:
- load faster
- resist inadvertent changes
- Testing verification one module at time.

Points to be noted

- 90% of processor time is spend in executing 10% of code. Identify this 10% .
- Listen to customer while developing specification
- Prototype complex task on host computer and investigate their behavior.
- Design architecture for debugging and testing.
- Code small modules that you can test and forget
- Code a single entry and exit in routines
- Document and comment carefully

Coupling And Cohesion

- Modules should have minimum communication or coupling
- If two tasks/ process communicate heavily they should reside in same module .
- Cohesion means that everything within module should be closely related
- Modules should have maximum cohesion

Documentation And Source Control

- Documentation : first to begin and last to finish to ensure completeness and veracity.
- Back up source files: disks, CD, tape drivers
- Store multiple copies in separate location
- File storage is cheap but reconstructing lost data is expensive and impossible.

Scheduling

- Should record all efforts expended in current jobs to estimate future job
- Timing of meeting, planning, designing, debugging testing should be properly planned
- Give enough time to debugging and testing
- Make allowances for the unexpected

9.7. User Interface

- A user interface is the system by which people (users) interact with a machine. The user interface includes hardware (physical) and software (logical) components. User interfaces exist for various systems, and provide a means of:
  - Input, allowing the users to manipulate a system
  - Output, allowing the system to indicate the effects of the users' manipulation

  Types:
  - **Graphical user interfaces** (GUI) accept input via devices such as computer keyboard and mouse and provide articulated graphical output on the computer monitor.
  - **Command line interfaces**, where the user provides the input by typing a command string with the computer keyboard and the system provides output by printing text on the computer monitor.
  - **Touch user interface** are graphical user interfaces using a touchpad or touch screen display as a combined input and output device.

9.8.1. Embedded Software

- **Embedded software** is computer software that plays an integral role in the electronics it is supplied with.
- Embedded software's principal role is not Information technology, but rather the interaction with the physical world. It's written for machines that are not, first and foremost, computers. Embedded software is 'built in' to the electronics in cars, telephones, audio equipment, robots, appliances, toys, security systems, pacemakers, televisions and digital watches, for example. This software can become very sophisticated in applications like airplanes, missiles, process control systems, and so on.
- Embedded software is usually written for special purpose hardware: that is computer chips that are different from general purpose CPUs, sometimes using Real-time operating system such as Linux (with patched kernel).

9.8.2. Real Time Software

- Real Time System is the study of hardware and software systems that are subject to a "real-time constraint"— e.g. operational deadlines from event to system response. Real-time programs must guarantee response within strict time constraints.
- A real-time software is one where the correct functioning of the system depends on the results produced by the system and the time at which these results are produced.
- A soft real-time system is a system whose operation is degraded if results are not produced according to the specified timing requirements.
- A hard real-time system is a system whose operation is incorrect if results are not produced according to the timing specification.
- Real-time software may use one or more of the following: synchronous programming languages, real-time operating systems, and real-time networks, each of which provide essential frameworks on which to build a real-time software application.