Name: KARTHIK V

TEJRAJ LINGADHAL

SUDHEER BIDARHALLI

DEVENDRA.B.HUPRI

Email ID: karhik543v@gmail.com, teja.laugh@gmail.com, sudheerinbloom@gmail.com, dpkhupri@gmail.com

Contact No.: 7411814804

9742352335

Stream Code: CPT

Branch: Information Science & Engg.

College: BV Bhoomaraddi College of Engg & Tech., Hubli

SOFTWARE ENGINEERING

KARTHIK V1, TEJRAJ LINGADHAL2, SUDHEER BIDARHALLI3, DEVENDRA.B.HUPRI4

*Information Science & Engineering,*

*B.V.B college of engineering Hubli*

karhik543v@gmail.com1, teja.laugh@gmail.com2, sudheerinbloom@gmail.com3, dpkhupri@gmail.com4

***Abstract*—This paper describes how to incorporate the above phases effectively through different process models so as to maintain the quality and productivity of software. The efficient software is mainly dependent on the above said phases which have to be carried out in a appropriate manner by maintaining cost, schedule and quality.**

1. INTRODUCTION

Software Engineering is the systematic approach to development, operation, maintenance, and retirement of software. Software is a collection of programs, procedures, rules, and associated documentation and data (IEEE).

Software is important because it is used by a great many people in society. Software engineers have a moral and ethical responsibility to ensure that the software they design does no serious harm to any people. Software engineers tend to be concerned with the technical elegance of their software products. Customers tend to be concerned only with whether or not a software product meets their needs and is easy to use.

There are two categories of software’s they are:

 Industrial Strength Software: The software which is built for the use of the customers and deployed in the market.

 Student Strength Software: The software which is built by the student for his purpose of academics.

The problem domain for software engineering is industrial strength software. Software engineering aims to provide methods for systematically developing (industrial strength) software Besides developing software the goal is to achieve high quality and productivity (Q&P). Methods used must accommodate changes, and must be able to handle large problems. Quality and Productivity is an essential goal and mainly depends on people, process, and technology. Process is a sequence of steps performed to achieve some goal. Processes help people become more productive and create fewer errors. Tools help people execute some tasks in the process more efficiently and effectively which intern makes process form the core. A software project is one instance of the development problem. Development process takes the project from user needs to software. Software Process is the sequence of steps performed to produce software with high quality, within budget and schedule. Two major processes are:

Development – focuses on development and quality steps, needed to engineer the software.

Project management – focuses on planning and controlling the development process.

Project is to build a software system within cost and schedule and with high quality which satisfies the customer. A process model specifies a general process that is optimal for a class of problems. A project may select its process using one of the process models. The process models have the common phases they are REQUIREMENTS ANALYSIS, ARCHITECTURE, DESIGN, CODING, TESTING, MAINTENANCE.

*A.* The Evolving Role of Software

The main point of this section is that the primary purpose of software is that of information transformer. Software is used to produce, manage, acquire, modify, display, and transmit information anywhere in the world. The days of the lone programmer are gone. Modern software is developed by teams of software specialists. Yet, the software developer's concerns have remained the same. Why does software take so long to complete? Why does it cost so much to produce? Why can't all errors be found and removed before software is delivered to the customer?

*B.* Software

Software is not like the artifacts produced in most other engineering disciplines. Software is developed it is not manufactured in the classical sense. Building a software product is more like constructing a design prototype. Opportunities for replication without customization are not very common. Software may become deteriorate, but it does not wear out. The chief reason for software deterioration is that many changes are made to a software product over its lifetime. As changes are made, defects may be inadvertently introduced to other portions of the software that interact with the portion that was changed.

*C.* Software: A Crisis on the Horizon

Many people have written about the "software development crisis". There seems to be an inordinate fascination with the spectacular software failures and a tendency to ignore the large number of successes achieved by the software industry. Software developers are capable of producing software the functions properly most of the time. The biggest problem facing modern software engineers is trying to figure out how to produce software fast enough to meet the growing demand for more products, while also having to maintain a growing volume of existing software.

*D*. Software Myths

The myths presented in this section provide a good source of material for class discussion. Managers need to understand that buying hardware and adding people does not spontaneously solve all software development problems. Customers need to understand that computer software does not make all their other problems go away. If the customer can not define his or her needs clearly, it is not possible to build a software product to meet these needs. If a customer does not have defined business processes without computer support, building computer software will not create these processes automatically. Software engineers must be committed to the concept of doing things right the first time. Software quality does not happen on its own after a product is finished. Quality must be built into every portion of the software development process.

1. THE PROCESS

This chapter discusses several process models used by professional software developers to manage large-scale software projects. No software process works well for every project. However, every project needs to conform to some systematic process in order to manage software development activities that can easily get out of control. Software processes help to organize the work products that need to be produced during a software development project. Ultimately the best indicator of how well a software process has worked is the quality of the deliverables produced. A well-managed process will produce high quality products on time and under budget.

*A.* Software Engineering

A layered Technology Software engineering encompasses a process, the management of activities, technical methods, and use of tools to develop software products. Software is engineered by applying three distinct phases (definition, development, and support). Students need to understand that maintenance involves more than fixing bugs.

*B***.** The Software Process

This section discusses the concept of a software process framework and provides a brief overview of the Software Engineering Institute Capability Maturity Model. It is important to emphasize that the Capability Maturity Model does not specify what process model needs to be used for a given project or organization. Rather, it provides a means of assessing how well an organization's processes allow them to complete and manage new software projects.

*C*. Software Process Models

The terms "software process model" and "software engineering paradigm" are used interchangeably in the literature. This chapter presents overviews of several software process models. It is easy for students to become so lost in the details of the various process models that they fail to see the features the models have in common with each other. Another difficulty students have is their belief that each phase of a process is performed completely independently of the other phases. The reality is that there tends to be lots overlap among the phases.

*D*. The Linear Sequential Model

The linear sequential model is also known as the "classic lifecycle" or "waterfall model". System development proceedsthough the phases (analysis, design, coding, testing, support) inorder. This is a good model to use when requirements are wellunderstood. If a phase must be revisited in this model, processfailure is indicated (more thorough requirements analysis isneeded).

*E*. The Prototyping Model

This model is good to use when the customer has legitimate needs, but is not able to articulate the details at the start of the project. A small mock-up of a working system is developed and presented to the customer. Sometimes this first system is discarded and sometimes it is extended based on the customer's feedback.

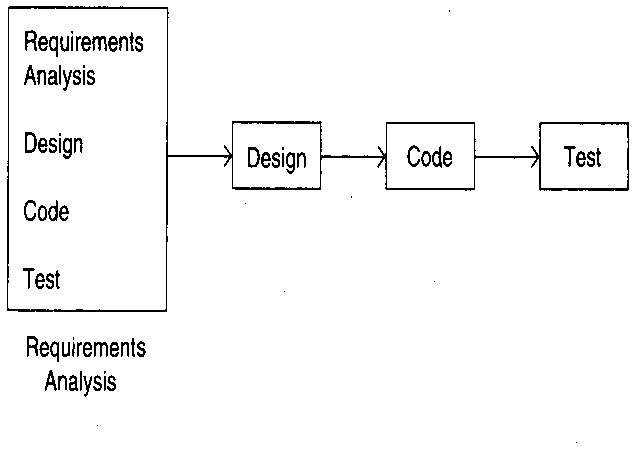


Fig. Describes the **prototyping model** of process models

*F.* Iterative Development

Counters the “all or nothing” drawback of the waterfall model (i.e the output/product is produced only at the end is solved in this model). Combines benefit of prototyping and waterfall. Develop and deliver software in increments. Each increment is complete in itself. Can be viewed as a sequence of waterfalls. Feedback from one iteration is used in the future iterations. First simple initial implementation is done for a subset of the overall problem. Each increment provide a feedback to the client that is useful for determining the final requirements of the system. A project control list is prepared which contain the tasks that must be performed to obtain the final implementation. Each iteration removes the next task from the list and implements the task which include design phase, implementation phase and analysis phase. Each iteration gives feedback to the next iteration. Selection of tasks in this manner will minimize the chances of error and reduce the redesign work. The process is iterated until the project control list is empty. The design and implementation phases of each step can be performed in a top-down manner or by using some other technique.

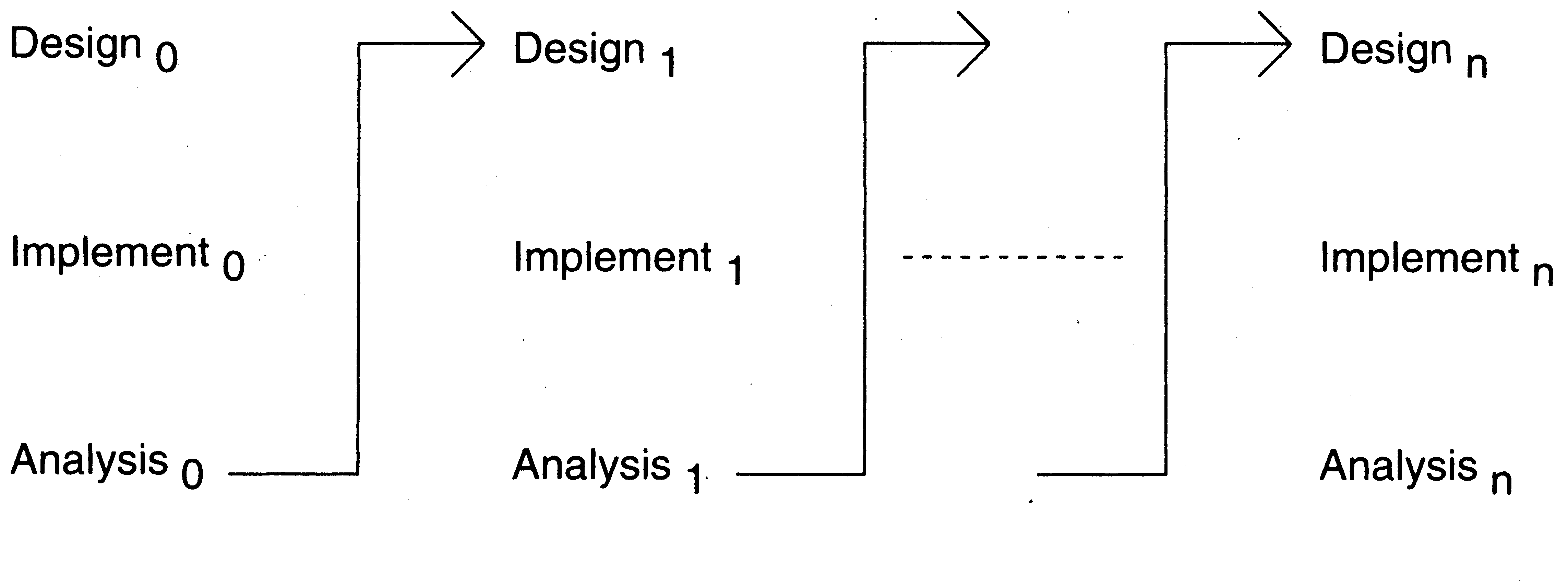


Fig. 2 Shows the process of the iterative model

G. Spiral development

Process is represented as a spiral rather than as a sequence of activities with backtracking. Each loop in the spiral represents a phase in the process. No fixed phases such as specification or design - loops in the spiral are chosen depending on what is required. Risks are explicitly assessed and resolved throughout the process. Inner most loop is concerned with the system feasibility, next loop with the requirement specification, the next loop with system design & so on.



Fig. 3 Shows the process spiral model

* Objective setting
  + Specific objectives for the phase are identified.
* Risk assessment and reduction
  + Risks are assessed and activities put in place to reduce the key risks.
* Development and validation
  + A development model for the system is chosen which can be any of the generic models.
* Planning
  + The project is reviewed and the next phase of the spiral is planned.

*F*. The RAD Model

The rapid application deployment model is a high-speed adaptation of the linear sequential model. Project requirements must be well understood and the project scope tightly constrained. Developers are often able to use component-based construction techniques to build a fully functional system in a short time period.

*G*. Waterfall Model

Phases are organized in Linear sequence of stages/phases. A phase starts only when the previous has completed; no feedback. The phases partition the project, each addressing a separate concern. Linear ordering implies each phase should have some output. The output must be verified and validated Outputs of earlier phases: work products. Planning usually overlaps with the requirements analysis. It has to be prepared before the other phase begins. Common outputs of a waterfall: SRS, project plan, design docs, test plan and reports, final code, software manuals.

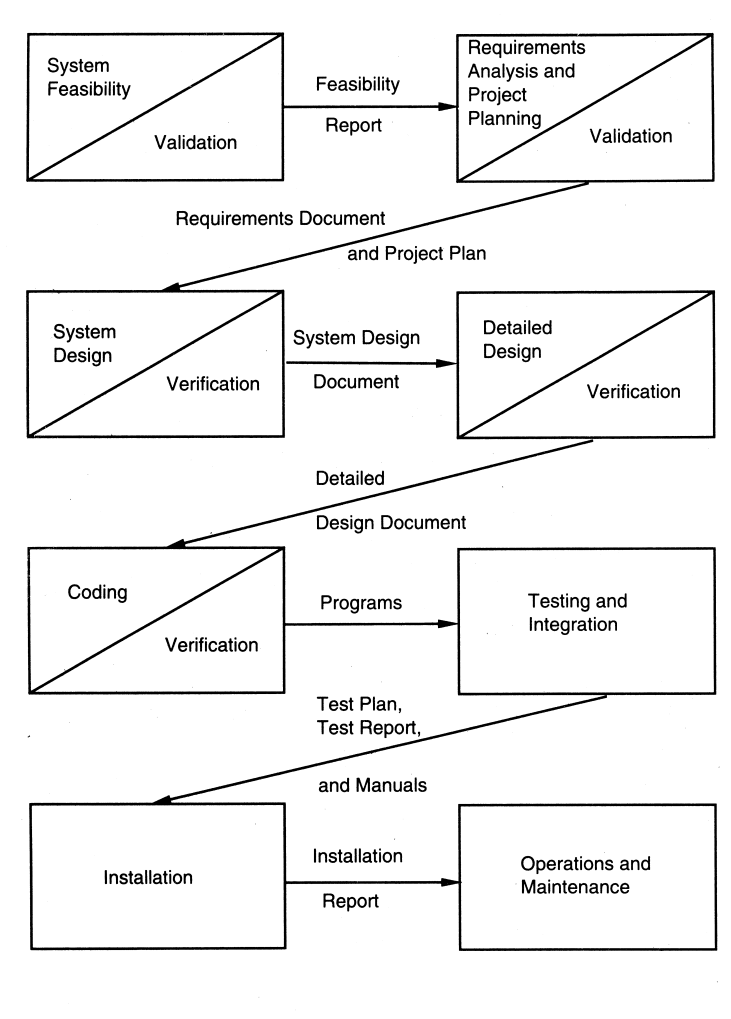


Fig. 4 Shows the process Water Fall Model

*.*

*G.* Evolutionary Models

The two most important models in this section are the incremental model and the spiral model. The incremental model combines elements of the linear sequential model applied repetitively with the iterative philosophy of the prototyping model. Each increment produces a working version of a software product with increasing functionality. There is no throwaway code. The spiral model also combines the iterative nature of prototyping with the systematic control found in the linear sequential model. An essential component of the spiral model is that assessment of both management and technical risks is performed as each incremental release is completed.

*H.* Component-Based Development

Object-based technologies provide the technical framework for component-based software engineering. The component-based development (CBD) model incorporates many of the iterative characteristics of the spiral model. The main difference is that in CBD the emphasis is on composing solutions from prepackaged software components or classes. This CBD emphasizes software reusability. The unified software development process is a example of CBD that has been proposed for industrial use. The unified modeling language (UML) is used to define the components and interfaces used in the unified software development process.

*I.* The Formal Methods Model

Formal methods in software development require the use ofrigorous mathematical notation to specify, design, and verify computer-based systems. Mathematical proof is used to verifythe correctness of a system (not empirical testing). Clean roomsoftware engineering is an example of this approach. Whileformal methods have the potential to produce defect-freesoftware, the development of formal models is both time consumingand expensive.

*J*. Fourth Generation Techniques

This is a broad class of techniques. The general idea is that a software tool is used to describe a system in manner understood by the customer using a specialized design language or graphical notation. In the ideal case, the tool then generates source code from this system description that can be compiled into a running system. The running system then needs to be tested and refined using other software engineering processes. There is some risk that the customer is not able to describe the system in sufficient detail or that the initial system will be deployed without sufficient testing. Section 2.12 uses a short essay by Margaret Davis to put process and product issues into perspective. If you’re teaching a graduate course, I’d recommend Phillip Howard’s *The Death of Common Sense,* as outside reading on the failures of a wholly process-oriented mind set.

1. REQUIREMNTS ANALYSIS

Software requirements analysis is intended to bridge the gaps between system requirements engineering and software design. In this chapter the focus of software requirements analysis is on the process of modeling the data, function, and performance of the software to be developed. The process model used has five areas of effort: problem recognition; evaluation and synthesis; modeling; specification; and review. It is important to remind students that requirements analysis is an important part of the software configuration management plan and the software quality assurance plan.

*B.* Requirements Elicitation for Software

This section discusses the task of software requirements elicitation. Two elicitation techniques are discussed (the use of context free questions in structured customer interviews and facilitated application specification techniques). Having students use these techniques to work with real customers or role-play working with simulated customers is a worthwhile task. If there is not enough time to do this in a one-semester course, if should be done in the senior projects course. Quality function deployment is described as a quality management technique that helps software engineers understand the relative importance of solution elements to the customer. Having students develop customer voice tables for their own projects and reviewing them with real or simulated customers would be a good lab activity. Encouraging students to develop use-cases as part of their requirements analysis will help prepare them for object-oriented analysis later in the text.

*B.* Analysis Principles

This section presents guidelines for conducting software requirements analysis. The perspective presented is that the goal of software requirements analysis is to build working models of the information, functional characteristics, and behavioral performance of a software product. It may be wise to focus on the general characteristics of models when lecturing on this chapter. Presenting examples of horizontal and vertical partitioning of data, function, and/or behavior might be good to do. It is important to convince students that going straight to the implementation view of software requirements, without considering the essential view, should not be done in the real world. Going directly to the implementation view is similar to deciding on a technology and then trying to define the customer's needs based on the capabilities of the technology (a strategy that is rarely satisfactory from the customer's perspective).

*C.* Software Prototyping

The pros and cons of using software prototyping during the analysis phase of the software process should be discussed. Students should not to be allowed to believe that prototypes are used to avoid writing the formal analysis and design documents. Prototypes are particularly good for working out the appearance details and behavioral characteristics of a software user interface. Techniques exist for developing models of computer-user interaction without building throwaway prototypes. Evolutionary prototyping is usually a more cost effective choice, if the user has a good understanding of his or her own goals and needs.

*D*. Specification

Students need to understand that is not generally possible tocompletely separate software requirements specification fromdesign. Even the waterfall model contains some provisions forrevisiting the specification phase if defects are uncoveredduring the design phase. Students should be encouraged to usethe software requirements specification template on the textweb site as they work on the their own course projects. Modelrepresentations and specification notations are discussed inmore detail in the next chapter. It is important for students toremember that the section on validation criteria is an importantpart of the specification document. Part of the reason for this, isthat the validation criteria along with the software models (data,functional, and behavioral) define a contract between thedeveloper and the client. When this contract is satisfied theproduct is finished and the developer can be paid.

*E. Specification Review*

The purpose of the specification review is to make sure that the customer and developer agree on details of the software requirements (or prototype) before beginning the major design work. This implies that both the customer and developer need to be present during the review meeting. Students will benefit from having to present their requirements specifications to a customer or to their classmates.

1. DESIGN CONCEPTS AND PRINCIPLES

The design model consists of the data design, architectural design, interface design, and component-level design. Although many of the design concepts and principles discussed in this chapter may have been presented in earlier courses, it is important to reemphasize each concept/principle so that all students have a consistent understanding of them. Students should be encouraged to use the design document template form the SEPA web site as a basis for the design documents they write for their own software projects.

*A.* Software Design and Software Engineering

It is important for students to understand the mapping from analysis model to the design model. The data design is derived from the data dictionary and the ERD. The architectural design is derived from the DFD. The interface design comes from the DFD and CFD. The component level design is derived from the PSPEC, CSPEC, and the STD. Students will need to be reminded that all design work products must be traceable to the software requirements document and that all design work products must be reviewed for quality.

*B .*The Design Process

This section makes the point that software design is an iterative process that is traceable to the software requirements analysis process. Students need to be reminded that many software projects iterate through the analysis and design phases several times. Pure separation of analysis and design may not always be possible or desirable. Having your students discuss the generic design guidelines, as a class may be a worthwhile activity.

*C.* Design Principles

This section presents several fundamental design principles. Students need to be aware of the rationale behind each principle.

*D*. Design Concepts

This sect ion discusses many significant design concepts (abstraction, refinement, modularity, software architecture, control hierarchy, structural partitioning, data structure, software procedure, information hiding). For some of these concepts a simple definition will be sufficient. For others (e.g. control hierarchy) presenting additional examples may be helpful.

*E.* Effective Modular Design

Several types of coupling and cohesion are described in this section. Students may have difficulty distinguishing the various types of coupling and cohesion without seeing examples of each. Students will have difficulty in remembering that high cohesion is good and that high coupling is bad. It may help to remind them that functional independence is the goal for each module. This is more likely when modules have single purposes (high cohesion) and rely on their own resources for data and control information (low coupling).

*F*. Design Heuristics for Effective Modularity

This section lists several fairly abstract heuristics for creating good program structure through careful module design. Discussing examples of programs that make use of these heuristics and some that violate these heuristics may be a good thing to do.

*G.* Design Documentation

This section lists the major sections of a software design specification document. A document template appears on the SEPA web site. It is important to get students in the habit of using diagrams to represent design information whenever it is feasible.

1. TESTING

Software testing is as old as the hills in the history of digital computers. The testing of software is an important means of assessing the software to determine its quality. Since testing typically consumes 40~50% of development efforts, and consumes more effort for systems that require higher levels of reliability, it is a significant part of the software engineering. With the development of Fourth generation languages (4GL), which speeds up the implementation process, the proportion of time devoted to testing increased. As the amount of maintenance and upgrade of existing systems grow, significant amount of testing will also be needed to verify systems after changes are made. Despite advances in formal methods and verification techniques, a system still needs to be tested before it is used. Testing remains the truly effective means to assure the quality of a software system of non-trivial complexity, as well as one of the most intricate and least understood areas in software engineering. Testing, an important research area within computer science is likely to become even more important in the future.

This retrospective on a fifty-year of software testing technique research examines the maturation of the software testing technique research by tracing the major research results that have contributed to the growth of this area. It also assesses the change of research paradigms over time by tracing the types of research questions and strategies used at various stages. We employ the technology maturation model given by Red wine and Riddle [15] as the framework of our studies of how the techniques of software testing first get the idea formulated, preliminarily used, developed, and then extended into a broader solution. Shaw gives a very good framework of software engineering research paradigms in [17], which classifies the research settings, research approaches, methods, and research validations that have been done by software researchers. Shaw’s model is used to evaluate the research strategies for testing techniques used in our paper.

*A.* The Taxonomy of Testing Techniques

Software testing is a very broad area, which involves many other technical and non-technical areas, such as specification, design and implementation, maintenance, process and management issues in software engineering. Our study focuses on the state of the art in testing techniques, as well as the latest techniques which representing the future direction of this area. Before stepping into any detail of the maturation study of these techniques, let us have a brief look at some technical concepts that are relative to our research.

*B.* The Goal of Testing

In different publications, the definition of testing varies according to the purpose, process, and level of testing described. Miller gives a good description of testing in. The general aim of testing is to affirm the quality of software systems by systematically exercising the software in carefully controlled circumstances. Miller’s description of testing views most software quality assurances activities as testing. He contends that testing should have the major intent of finding errors. A good test is one that has a high probability of finding an as yet undiscovered error, and a successful test is one that uncovers an as yet undiscovered error. This general category of software testing activities can be further divided. For purposes of this paper, testing is the dynamic analysis of a piece of software, requiring execution of the system to produce results, which are then compared to expected outputs.

*C.* The Testing Spectrum

Testing is involved in every stage of software life cycle, but the testing done at each level of software development is different in nature and has different objectives.

***Unit Testing*** is done at the lowest level. It tests the basic unit of software, which is the smallest testable piece of software, and is often called “unit”, “module”, or “component” interchangeably.

***Integration Testing*** is performed when two or more tested units are combined into a larger structure. The test is often done on both the interfaces between the components and the larger structure being constructed, if its quality property cannot be assessed from its components.

***System Testing*** tends to affirm the end-to-end quality of the entire system. System test is often based on the functional/requirement specification of the system. Non-functional quality attributes, such as reliability, security, and maintainability, are also checked.

***Acceptance Testing*** is done when the completed system is handed over from the developers to the customers or users. The purpose of acceptance testing is rather to give confidence that the system is working than to find errors.

*D*. Static Analysis and Dynamic Analysis

Based on whether the actual execution of software under evaluation is needed or not, there are two major categories of quality assurance activities:

***Static Analysis*** focuses on the range of methods that are used to determine or estimate software quality without reference to actual executions. Techniques in this area include code inspection, program analysis, symbolic analysis, and model checking.

***Dynamic Analysis*** deals with specific methods for ascertaining and/or approximating software quality through actual executions, i.e., with real data and under real (or simulated) circumstances. Techniques in this area include synthesis of inputs, the use of structurally dictated testing procedures, and the automation of testing environment generation. Generally the static and dynamic methods are sometimes inseparable, but can almost always discussed separately. In this paper, we mean dynamic analysis when we say testing, since most of the testing activities (thus all the techniques studied in this paper) require the execution of the software.

*E.* Functional Technique and Structural Technique

The information flow of testing is shown in Figure 1. As we can see, testing involves the configuration of proper inputs, execution of the software over the input, and the analysis of the output. The “Software Configuration” includes requirements specification, design specification, source code, and so on. The “Test Configuration” includes test cases, test plan and procedures, and testing tools. Based on the testing information flow, a **testing technique** specifies the strategy used in testing to select input test cases and analyze test results. Different techniques reveal different quality aspects of a software system, and there are two major categories of testing techniques, functional and structural.

***Functional Testing***: the software program or system under test is viewed as a “**black box**”. The selection of test cases for functional testing is based on the **requirement** or **design specification** of the software entity under test. Examples of expected results, some times are called **test oracles**, include 3 requirement/design specifications, hand calculated values, and simulated results. Functional testing emphasizes on the **external behavior** of the software entity.

***Structural Testing***: the software entity is viewed as a “**white box**”. The selection of test cases is based on the **implementation** of the software entity. The goal of selecting such test cases is to cause the execution of specific spots in the software entity, such as specific statements, program branches or paths. The expected results are evaluated on a set of **coverage criteria**. Examples of coverage criteria include path coverage, branch coverage, and data-flow coverage. Structural testing emphasizes on the **Internal structure** of the software entity.

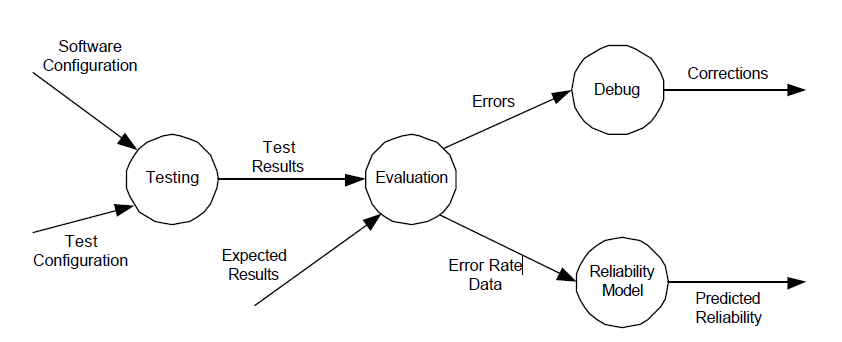


Fig. 4 Shows the process testing information flow

1. CONCLUSION

The ultimate goal of software testing is to help designers, developers, and managers construct systems with high quality. Thus research and development on testing aim at efficiently performing effective testing – to find more errors in requirement, design and implementation, and to increase confidence that the software has various qualities. Testing technique research leads to the destination of practical testing methods and 4 tools. Progress toward this destination requires fundamental research, and the creation, refinement, extension, and popularization of better methods. The standard of progress for the research of testing techniques include: Degree of acceptance of the technology inside and outside the research communityDegree of dependability on other areas of software engineering. Change of research paradigms in response to the maturation of software development technologies. Feasibility of techniques being used in a widespread practical scope, and Spread of technology – classes, trainings, management attention.

ACKNOWLEDGMENT

We have been bestowed the privilege of expressing our gratitude to everyone who helped us in completing the dissertation work.

First, we would like to thank our guide **Prof. SUNITHA HIREMATH.** Lecturer, Information Science & Engineering department, BVBCET - Hubli, whose guidance was great use to us during this paper presentation work.

We find our acknowledgement incomplete without thanking **Dr. Ashok Shettar**. Principal BVBCET, Hubli -31 for the inspiration and co-operation.

Finally we thank one and all who have directly and indirectly assisted us in the project work.

REFERENCES

**[1]** G. Bernet, L. Bouaziz, and P. LeGall, “A Theory of Probabilistic Functional Testing,” *Proceedings of the 1997 International Conference on Software Engineering*, 1997, pp. 216 –226

**[2]** B. Beizer, “Software Testing Techniques,” Second Edition, *Van Nostrand Reinhold Company Limited,* 1990, ISBN 0-442-20672-0

**[3]** S. Beydeda and V. Gruhn, “An integrated testing technique for component-based software,” *ACS/IEEE International Conference on Computer Systems and Applications*, June 2001, pp 328 – 334

**[4]** A. Bertolino, P. Inverardi, H. Muccini, and A. Rosetti, “An approach to integration testing based on architectural descriptions,” *Proceedings of the IEEE ICECCS-* 97, pp. 77-84

**[5]** J.B. Good Enough and S. L. Gerhart, “Toward a Theory of Test Data Selection,” *IEEE Transactions on Software Engineering*, June 1975, pp. 156-173