VIRTUAL UNIVERSITY OF PAKISTAN

Software EngineeringII

CS605

Course Handouts

GRADUP

Practice. Pass. Graduate.

thegradup.com

CS605
Software Engineering-II

Table of Content

Lecture No. 1 4
Introduction to Software Engineering 4
Lecture No. 2 12
Software Process 12
Lecture No. 3 14
Software Lifecycle Models 14
Lecture No. 4 18
Incremental Models 18
Lecture No. 5 23
Object-Oriented Lifecycle Models 23
Lecture No. 6 27
Software Project Management Concepts 27
Lecture No. 7 32
The Software Team 32
Lecture No. 8 35
The Project Management 35
Lecture No. 9 37
Software Size Estimation 37
Lecture No. 10 41
Function Point Counting Process 41
Lecture No. 11 46
Function Point Counting Process (cont.) 46
Lecture No. 12 65
Software Process and Project Metrics 65
Lecture No. 13 67
Software Quality Factors 67
Lecture No. 14 71
Metrics for specification quality 71
Lecture No. 15 75
Statistical Control Techniques – control charts 75
Lecture No. 16 78
Interpreting Measurements 78
Lecture No. 17 80
Software Project Planning 80
Lecture No. 18 84
Risk analysis and management 84
Lecture No. 19 86
Types of Risks 86
Lecture No. 20 89
Assessing Risk Impact 89
Lecture No. 21 92
Software Project Scheduling and Monitoring 92
Lecture No. 22 95
Relationship between people and effort 95
Lecture No. 23 99

Task Network **99**
Lecture No. 24 ..... **100**
Scheduling **100**
Lecture No. 25 ..... **102**
Project Tracking **102**
Lecture No. 26 ..... **104**
Error Tracking **104**
Lecture No. 27 ..... **106**
Software Quality Assurance **106**
Lecture No. 28 ..... **109**
Software Reviews **109**
Lecture No. 29 ..... **112**
Formal Technical Reviews **112**
Lecture No. 30 ..... **114**
Statistical Software Quality Assurance **114**
Lecture No. 31 ..... **117**
Software Safety **117**
Lecture No. 32 ..... **118**
Poka-Yoke (Mistake-Proofing) **118**
Lecture No. 33 ..... **119**
Software Configuration Management (SCM) **119**
Lecture No. 34 ..... **122**
Product Release Version Numbering System **122**
Lecture No. 35 ..... **124**
Change Control Process **124**
Lecture No. 36 ..... **127**
Requirement Management and CMM **127**
Lecture No. 37 ..... **132**
Measuring Change Activity **132**
Lecture No. 38 ..... **134**
Legacy systems **134**
Lecture No. 39 ..... **137**
Environment Assessment **137**
Lecture No. 40 ..... **140**
Forward Engineering **140**
Lecture No. 41 ..... **141**
Business Process Reengineering **141**
Lecture No. 42 ..... **142**
Software Refactoring **142**
Lecture No. 43 ..... **153**
Moving the amount calculation **153**
Lecture No. 44 ..... **168**
Capability Maturity Model Integration (CMMI) **168**
Lecture No. 45 ..... **172**
CMM Overview **172**

Lecture No. 1

Introduction to Software Engineering

This course is a continuation of the first course on Software Engineering. In order to set the context of our discussion, let us first look at some of the definitions of software engineering.

Software Engineering is the set of processes and tools to develop software. *Software Engineering is the combination of all the tools, techniques, and processes that used in software production.* Therefore Software Engineering encompasses all those things that are used in software production like:

- Programming Language
- Programming Language Design
- Software Design Techniques
- Tools
- Testing
- Maintenance
- Development etc.

So all those thing that are related to software are also related to software engineering.

Some of you might have thought that how programming language design could be related to software engineering. If you look more closely at the software engineering definitions described above then you will definitely see that software engineering is related to all those things that are helpful in software development. So is the case with programming language design. Programming language design is one of the major successes in last fifty years. The design of Ada language was considered as the considerable effort in software engineering.

These days object-oriented programming is widely being used. If programming languages will not support object-orientation then it will be very difficult to implement object-oriented design using object-oriented principles. All these efforts made the basis of software engineering.

Well-Engineered Software

Let's talk something about what is well-engineered software. Well-engineered software is one that has the following characteristics.

- It is reliable
- It has good user-interface
- It has acceptable performance
- It is of good quality
- It is cost-effective

Every company can build software with unlimited resources but well-engineered software is one that conforms to all characteristics listed above.

Software has very close relationship with economics. When ever we talk about engineering systems we always first analyze whether this is economically feasible or not. Therefore you have to engineer all the activities of software development while keeping its economical feasibility intact.

The major challenges for a software engineer is that he has to build software within limited time and budget in a cost-effective way and with good quality

Therefore well-engineered software has the following characteristics.

- Provides the required functionality
- Maintainable
- Reliable
- Efficient
- User-friendly
- Cost-effective

But most of the times software engineers ends up in conflict among all these goals. It is also a big challenge for a software engineer to resolve all these conflicts.

The Balancing Act!

Software Engineering is actually the balancing act. You have to balance many things like cost, user friendliness, Efficiency, Reliability etc. You have to analyze which one is the more important feature for your software is it reliability, efficiency, user friendliness or something else. There is always a trade-off among all these requirements of software. It may be the case that if you try to make it more user-friendly then the efficiency may suffer. And if you try to make it more cost-effective then reliability may suffer. Therefore there is always a trade-off between these characteristics of software.

These requirements may be conflicting. For example, there may be tension among the following:

- Cost vs. Efficiency
- Cost vs. Reliability
- Efficiency vs. User-interface

A Software Engineer is required to analyze these conflicting entities and tries to strike a balance.

Challenge is to balance these requirements.

Software Engineers always confront with the challenge to make a good balance of all these tings depending on the requirements of the particular software system at hand. He should analyze how much weight should all these things get such that it will have acceptable quality, acceptable performance and will have acceptable user-interface.

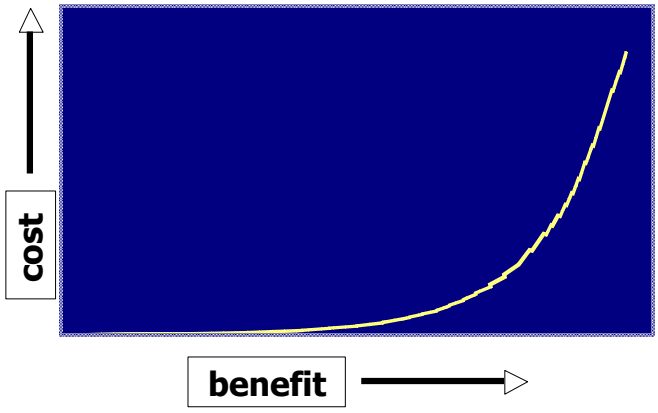
In some software the efficiency is more important and desirable. For example if we talk about a cruise missile or a nuclear reactor controller that are droved by the software systems then performance and reliability is far more important than the cost-effectiveness and user-friendliness. In these cases if your software does not react within a certain amount of time then it may result in the disaster like Chernobyl accident.

Therefore software development is a process of balancing among different characteristics of software described in the previous section. And it is an art to come up with such a good balance and that art can be learned from experience.

Law of diminishing returns

In order to understand this concept lets take a look at an example. Most of you have noticed that if you dissolve sugar in a glass of water then the sweetness of water will increase gradually. But at a certain level of saturation no more sugar will dissolved into water. Therefore at that point of saturation the sweetness of water will not increase even if you add more sugar into it.

The law of diminishing act describes the same phenomenon. Similar is the case with software engineering. Whenever you perform any task like improving the efficiency of the system, try to improve its quality or user friendliness then all these things involves an element of cost. If the quality of your system is not acceptable then with the investment of little money it could be improved to a higher degree. But after reaching at a certain level of quality the return on investment on the system's quality will become reduced. Meaning that the return on investment on quality of software will be less than the effort or money we invest. Therefore, in most of the cases, after reaching at a reasonable level of quality we do not try to improve the quality of software any further. This phenomenon is shown in the figure below.



Software Background

Caper Jones a renounced practitioner and researcher in the filed of Software Engineering, had made immense research in software team productivity, software quality, software cost factors and other fields relate to software engineering. He made a company named

Software Productivity Research in which they analyzed many projects and published the results in the form of books. Let's look at the summary of these results.

He divided software related activities into about twenty-five different categories listed in the table below. They have analyzed around 10000 software projects to come up with such a categorization. But here to cut down the discussion we will only describe nine of them that are listed below.

- Project Management
- Requirement Engineering
- Design
- Coding
- Testing
- Software Quality Assurance
- Software Configuration Management
- Software Integration and
- Rest of the activities

One thing to note here is that you cannot say that anyone of these activities is dominant among others in terms of effort putted into it. Here the point that we want to emphasize is that, though coding is very important but it is not more than 13-14% of the whole effort of software development.

Fred Brook is a renowned software engineer; he wrote a great book related to software engineering named "A Mythical Man Month". He combined all his articles in this book. Here we will discuss one of his articles named "No Silver Bullet" which he included in the book.

An excerpt from "No Silver Bullet" – Fred Brooks

> *Of all the monsters that fill the nightmares of our folklore, none terrify more than werewolves, because they transform unexpectedly from the familiar into horrors. For these we seek bullets of silver that can magically lay them to rest. The familiar software project has something of this character (at least as seen by the non-technical manager), usually innocent and straight forward, but capable of becoming a monster of missed schedules, blown budgets, and flawed projects. So we hear desperate cries for a silver bullet, something to make software costs drop as rapidly as computer hardware costs do. Scepticism is not pessimism, however. Although we see no startling breakthroughs, and indeed, such to be inconsistent with the nature of the software, many encouraging innovations are under way. A disciplined, consistent effort to develop, propagate and exploit them should indeed yield an order of magnitude improvement. There is no royal road, but there is a road. The first step towards the management of disease was replacement of demon theories and humours theories by the germ theory. The very first step, the beginning of hope, in itself dashed all hopes of magical solutions. It told workers that progress would be made stepwise, at great effort, and that a persistent, unremitting care would have to be paid to a discipline of cleanliness. So it is with software engineering today.*

So, according to Fred Brook, in the eye of an unsophisticated manager software is like a giant. Sometimes it reveals as an unscheduled delay and sometimes it shows up in the

form of cost overrun. To kill this giant the managers look for magical solutions. But unfortunately magic is not a reality. We do not have any magic to defeat this giant. There is only one solution and that is to follow a disciplined approach to build software. We can defeat the giant named software by using disciplined and engineered approach towards software development.

Therefore, *Software Engineering is nothing but a disciplined and systematic approach to software development*.

Now we will look at some of the activities involved in the course of software development. The activities involved in software development can broadly be divided into two major categories first is construction and second is management.

Software Development

The construction activities are those that are directly related to the construction or development of the software. While the management activities are those that complement the process of construction in order to perform construction activities smoothly and effectively. A greater detail of the activities involved in the construction and management categories is presented below.

Construction

The construction activities are those that directly related to the development of software, e.g. gathering the requirements of the software, develop design, implement and test the software etc. Some of the major construction activities are listed below.

- Requirement Gathering
- Design Development
- Coding
- Testing

Management

Management activities are kind of umbrella activities that are used to smoothly and successfully perform the construction activities e.g. project planning, software quality assurance etc. Some of the major management activities are listed below.

- Project Planning and Management
- Configuration Management
- Software Quality Assurance
- Installation and Training

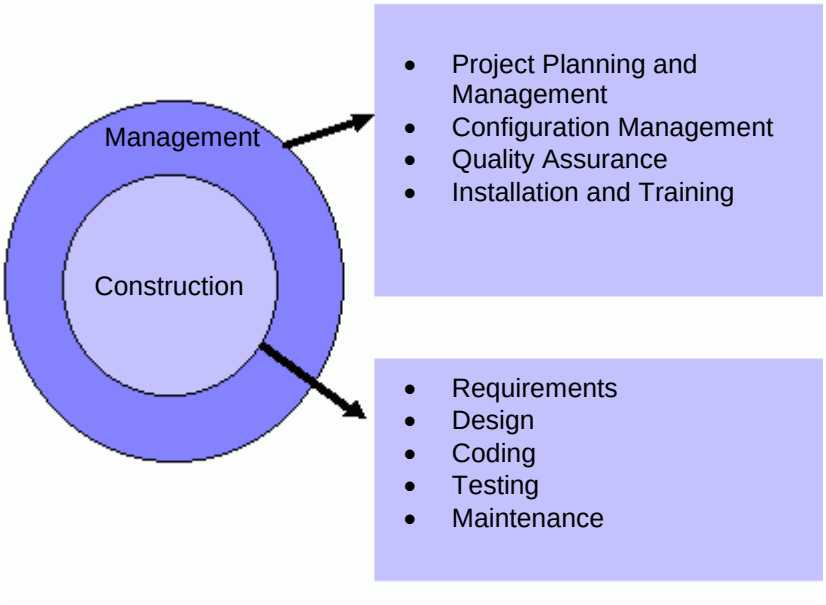


Figure1
Development Activities

As we have said earlier that management activities are kind of umbrella activities that surround the construction activities so that the construction process may proceed smoothly. This fact is empathized in the Figure1. The figure shows that construction is surrounded by management activities. That is, certain processes and rules govern all construction activities. These processes and rules are related to the management of the construction activities and not the construction itself.

A Software Engineering Framework

The software development organization must have special focus on quality while performing the software engineering activities. Based on this commitment to quality by the organization, a software engineering framework is proposed that is shown in Figure 2. The major components of this framework are described below.

Quality Focus: As we have said earlier, the given framework is based on the organizational commitment to quality. The quality focus demands that processes be defined for rational and timely development of software. And quality should be emphasized while executing these processes.

Processes: The processes are set of key process areas (KPAs) for effectively manage and deliver quality software in a cost effective manner. The processes define the tasks to be performed and the order in which they are to be performed. Every task has some deliverables and every deliverable should be delivered at a particular milestone.

Methods: Methods provide the technical "how-to's" to carryout these tasks. There could be more than one technique to perform a task and different techniques could be used in different situations.

Tools: Tools provide automated or semi-automated support for software processes, methods, and quality control.

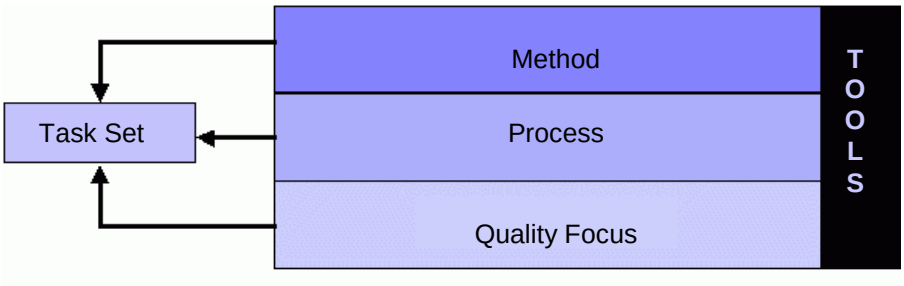


Figure 2
Software Engineering Framework

Software Development Loop

Let's now look at software engineering activities from a different perspective. Software development activities could be performed in a cyclic and that cycle is called software development loop which is shown in Figure3. The major stages of software development loop are described below.

Problem Definition: In this stage we determine what is the problem against which we are going to develop software. Here we try to completely comprehend the issues and requirements of the software system to build.

Technical Development: In this stage we try to find the solution of the problem on technical grounds and base our actual implementation on it. This is the stage where a new system is actually developed that solves the problem defined in the first stage.

Solution Integration: If there are already developed system(s) available with which our new system has to interact then those systems should also be the part of our new system. All those existing system(s) integrate with our new system at this stage.

Status Quo: After going through the previous three stages successfully, when we actually deployed the new system at the user site then that situation is called status quo. But once we get new requirements then we need to change the status quo.

After getting new requirements we perform all the steps in the software development loop again. The software developed through this process has the property that this could be evolved and integrated easily with the existing systems.

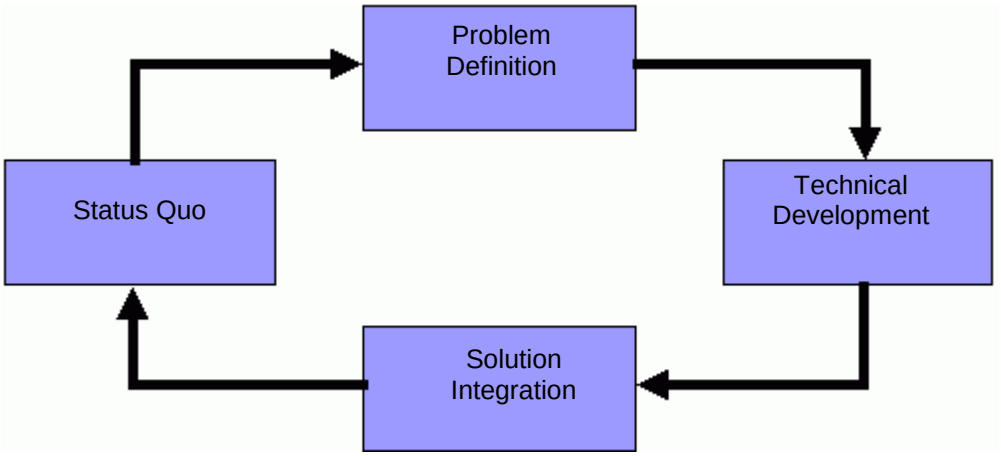


Figure3
Software Development Loop

Overview of the course contents

In the first course we studied the technical processes of software development to build industrial strength software. That includes requirement gathering and analysis, software design, coding, testing, and debugging. In this course our focus will be on the second part of Software Engineering, that is, the activities related to managing the technical development. This course will therefore include the following topics:

1. Software development process
2. Software process models
3. Project Management Concepts
4. Software Project Planning
5. Risk Analysis and Management
6. Project Schedules and Tracking
7. Software Quality Assurance
8. Software Configuration Management
9. Software Process and Project Metrics
10. Requirement Engineering Processes
11. Verification and Validation
12. Process Improvement
13. Legacy Systems
14. Software Change
15. Software Re-engineering

© Virtual University of Pakistan
thegradup.com

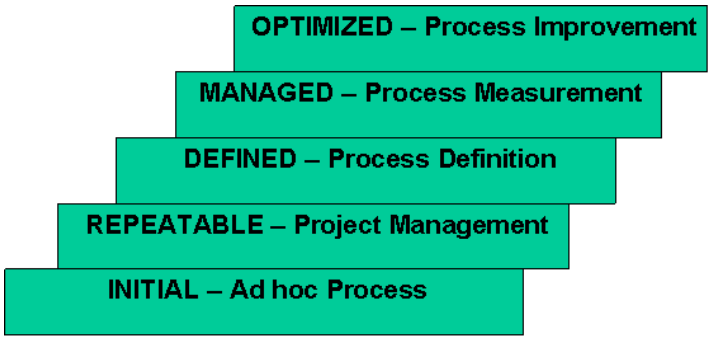
Lecture No. 2

Software Process

A software process is a road map that helps you create a timely, high quality result. It is the way we produce software and it provides stability and control. Each process defines certain deliverables known as the work products. These include programs, documents, and data produced as a consequence of the software engineering activities.

Process Maturity and CMM

The Software Engineering Institute (SEI) has developed a framework to judge the process maturity level of an organization. This framework is known as the Capability Maturity Model (CMM). This framework has 5 different levels and an organization is placed into one of these 5 levels. The following figure shows the CMM framework.



These levels are briefly described as follows"

1. Level 1 – Initial: The software process is characterized as ad hoc and occasionally even chaotic. Few processes are defined, and success depends upon individual effort. By default every organization would be at level 1.
2. Level 2 – Repeatable: Basic project management processes are established to track cost, schedule, and functionality. The necessary project discipline is in place to repeat earlier successes on projects with similar applications.
3. Level 3 – Defined: The software process for both management and engineering activities is documented, standardized, and integrated into an organizational software process. All projects use a documented and approved version of the organization's process for developing and supporting software.
4. Level 4 – Managed: Detailed measures for software process and product quality are controlled. Both the software process and products are quantitatively understood and controlled using detailed measures.
5. Level 5 – Optimizing: Continuous process improvement is enabled by qualitative feedback from the process and from testing innovative ideas and technologies.

SEI has associated key process areas with each maturity level. The KPAs describe those software engineering functions that must be present to satisfy good practice at a particular level. Each KPA is described by identifying the following characteristics:

1. Goals: the overall objectives that the KPA must achieve.
2. Commitments: requirements imposed on the organization that must be met to achieve the goals or provide proof of intent to comply with the goals.
3. Abilities: those things that must be in place – organizationally and technically – to enable the organization to meet the commitments.
4. Activities: the specific tasks required to achieve the KPA function
5. Methods for monitoring implementation: the manner in which the activities are monitored as they are put into place.
6. Methods for verifying implementation: the manner in which proper practice for the KPA can be verified.

Each of the KPA is defined by a set of practices that contribute to satisfying its goals. The key practices are policies, procedures, and activities that must occur before a key process area has been fully instituted.

The following table summarizes the KPAs defined for each level.

Level	KPAs
1	No KPA is defined as organizations at this level follow ad-hoc processes
2	• Software Configuration Management • Software Quality Assurance • Software subcontract Management • Software project tracking and oversight • Software project planning • Requirement management
3	• Peer reviews • Inter-group coordination • Software product Engineering • Integrated software management • Training program • Organization process management • Organization process focus
4	• Software quality management • Quantitative process management
5	• Process change management • Technology change management • Defect prevention

Lecture No. 3

Software Lifecycle Models

Recalling from our first course, a software system passes through the following phases:

1. Vision – focus on *why*
2. Definition – focus on *what*
3. Development – focus on *how*
4. Maintenance – focus on *change*

During these phases, a number of activities are performed. A lifecycle model is a series of steps through which the product progresses. These include requirements phase, specification phase, design phase, implementation phase, integration phase, maintenance phase, and retirement. Software Development Lifecycle Models depict the way you organize your activities.

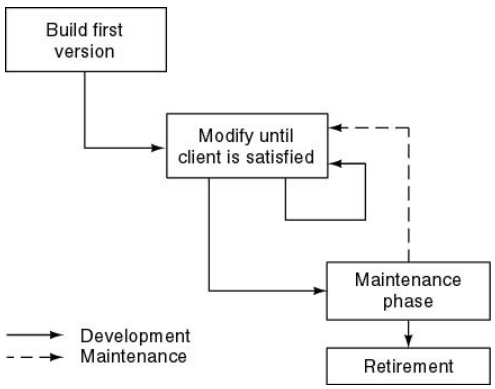
There are a number of Software Development Lifecycle Models, each having its strengths and weaknesses and suitable in different situations and project types. The list of models includes the following:

- Build-and-fix model
- Waterfall model
- Rapid prototyping model
- Incremental model
- Extreme programming
- Synchronize-and-stabilize model
- Spiral model
- Object-oriented life-cycle models

In the following sections we shall study these models in detail and discuss their strengths and weaknesses.

Build and Fix Model

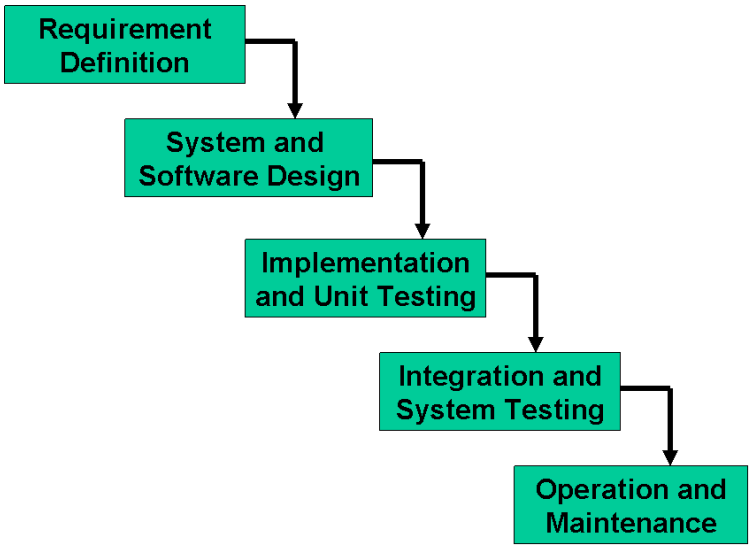
This model is depicted in the following diagram:



It is unfortunate that many products are developed using what is known as the build-and-fix model. In this model the product is constructed without specification or any attempt at design. The developers simply build a product that is reworked as many times as necessary to satisfy the client. This model may work for small projects but is totally unsatisfactory for products of any reasonable size. The cost of build-and fix is actually far greater than the cost of properly specified and carefully designed product.
Maintenance of the product can be extremely in the absence of any documentation.

Waterfall Model

The first published model of the software development process was derived from other engineering processes. Because of the cascade from one phase to another, this model is known as the waterfall model. This model is also known as linear sequential model. This model is depicted in the following diagram.



The principal stages of the model map directly onto fundamental development activities.

It suggests a systematic, sequential approach to software development that begins at the system level and progresses through the analysis, design, coding, testing, and maintenance.

In the literature, people have identified from 5 to 8 stages of software development.

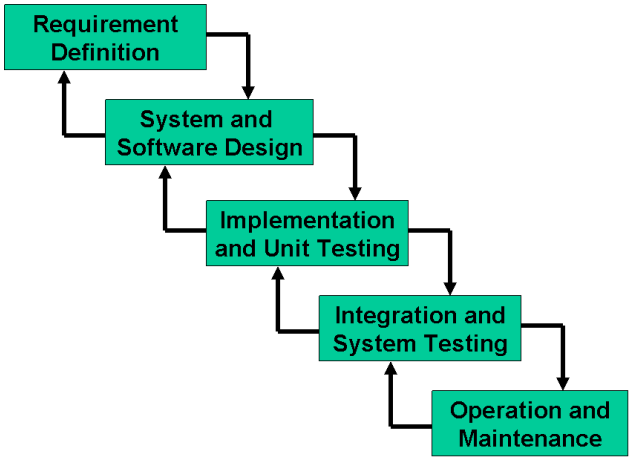
The five stages above are as follows:

1. Requirement Analysis and Definition: What - The systems services, constraints and goals are established by consultation with system users. They are then defined in detail and serve as a system specification.
2. System and Software Design: How – The system design process partitions the requirements to either hardware of software systems. It establishes and overall system architecture. Software design involves fundamental system abstractions and their relationships.

3. Implementation and Unit Testing: - How – During this stage the software design is realized as a set of programs or program units. Unit testing involves verifying that each unit meets its specifications.
4. Integration and system testing: The individual program unit or programs are integrated and tested as a complete system to ensure that the software requirements have been met. After testing, the software system is delivered to the customer.
5. Operation and Maintenance: Normally this is the longest phase of the software life cycle. The system is installed and put into practical use. Maintenance involves correcting errors which were not discovered in earlier stages of the life-cycle, improving the implementation of system units and enhancing the system's services as new requirements are discovered.

In principle, the result of each phase is one or more documents which are approved. No phase is complete until the documentation for that phase has been completed and products of that phase have been approved. The following phase should not start until the previous phase has finished.

Real projects rarely follow the sequential flow that the model proposes. In general these phases overlap and feed information to each other. Hence there should be an element of iteration and feedback. A mistake caught any stage should be referred back to the source and all the subsequent stages need to be revisited and corresponding documents should be updated accordingly. This feedback path is shown in the following diagram.



Because of the costs of producing and approving documents, iterations are costly and require significant rework.

The Waterfall Model is a documentation-driven model. It therefore generates complete and comprehensive documentation and hence makes the maintenance task much easier. It however suffers from the fact that the client feedback is received when the product is finally delivered and hence any errors in the requirement specification are not discovered until the product is sent to the client after completion. This therefore has major time and cost related consequences.

Rapid Prototyping Model

The Rapid Prototyping Model is used to overcome issues related to understanding and capturing of user requirements. In this model a mock-up application is created "rapidly" to solicit feedback from the user. Once the user requirements are captured in the prototype to the satisfaction of the user, a proper requirement specification document is developed and the product is developed from scratch.

An essential aspect of rapid prototype is embedded in the word "rapid". The developer should endeavour to construct the prototype as quickly as possible to speedup the software development process. It must always be kept in mind that the sole purpose of the rapid prototype is to capture the client's needs; once this has been determined, the rapid prototype is effectively discarded. For this reason, the internal structure of the rapid prototype is not relevant.

Integrating the Waterfall and Rapid Prototyping Models

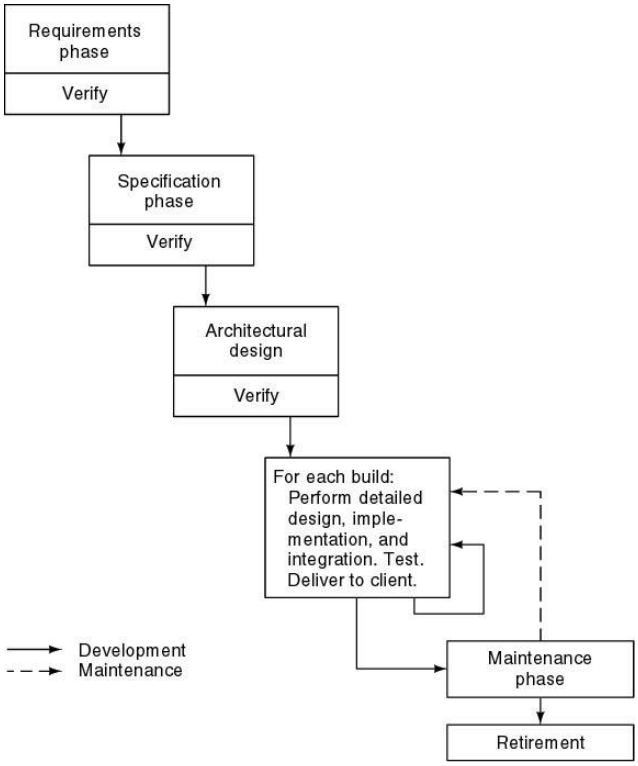
Despite the many successes of the waterfall model, it has a major drawback in that the delivered product may not fulfil the client's needs. One solution to this is to combine rapid prototyping with the waterfall model. In this approach, rapid prototyping can be used as a requirement gathering technique which would then be followed by the activities performed in the waterfall model.

Lecture No. 4

Incremental Models

As discussed above, the major drawbacks of the waterfall model are due to the fact that the entire product is developed and delivered to the client in one package. This results in delayed feedback from the client. Because of the long elapsed time, a huge new investment of time and money may be required to fix any errors of omission or commission or to accommodate any new requirements cropping up during this period. This may render the product as unusable. Incremental model may be used to overcome these issues.

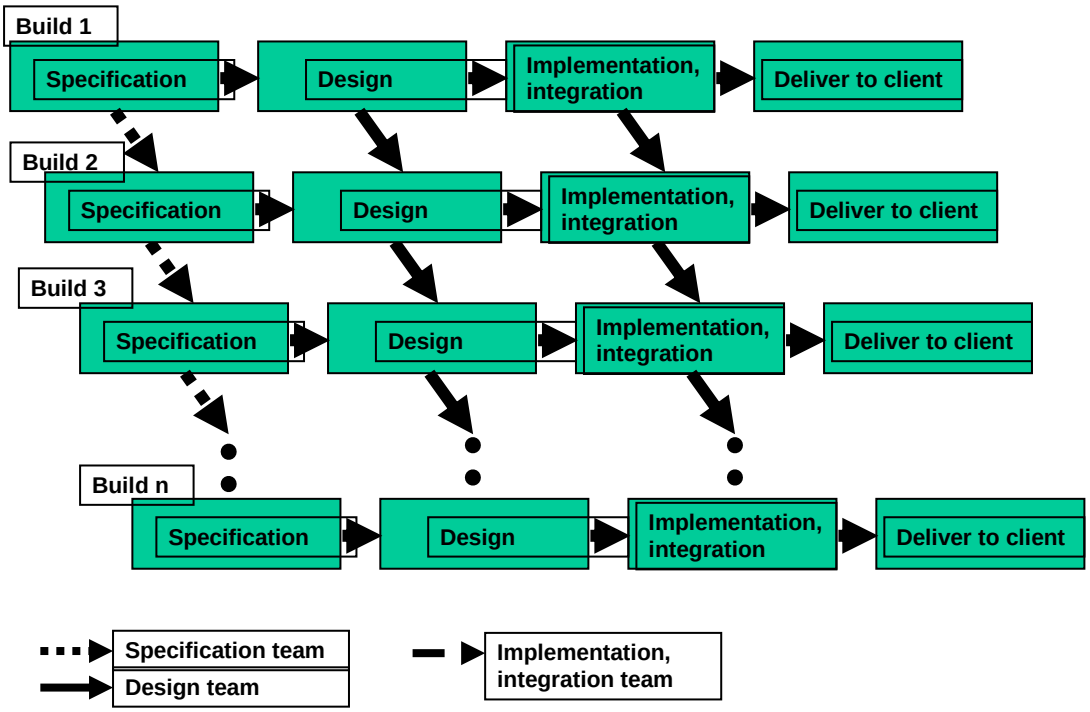
In the incremental models, as opposed to the waterfall model, the product is partitioned into smaller pieces which are then built and delivered to the client in increments at regular intervals. Since each piece is much smaller than the whole, it can be built and sent to the client quickly. This results in quick feedback from the client and any requirement related errors or changes can be incorporated at a much lesser cost. It is therefore less traumatic as compared to the waterfall model. It also required smaller capital outlay and yield a



rapid return on investment. However, this model needs and open architecture to allow integration of subsequent builds to yield the bigger product. A number of variations are used in object-oriented life cycle models.

There are two fundamental approaches to the incremental development. In the first case, the requirements, specifications, and architectural design for the whole product are completed before implementation of the various builds commences.

In a more risky version, once the user requirements have been elicited, the specifications of the first build are drawn up. When this has been completed, the specification team



turns to the specification of the second build while the design team designs the first build. Thus the various builds are constructed in parallel, with each team making use of the information gained in the all the previous builds.

This approach incurs the risk that the resulting build will not fit together and hence requires careful monitoring.

Rapid Application Development (RAD)

Rapid application development is another form of incremental model. It is a high speed adaptation of the linear sequential model in which fully functional system in a very short time (2-3 months). This model is only applicable in the projects where requirements are well understood and project scope is constrained. Because of this reason it is used primarily for information systems.

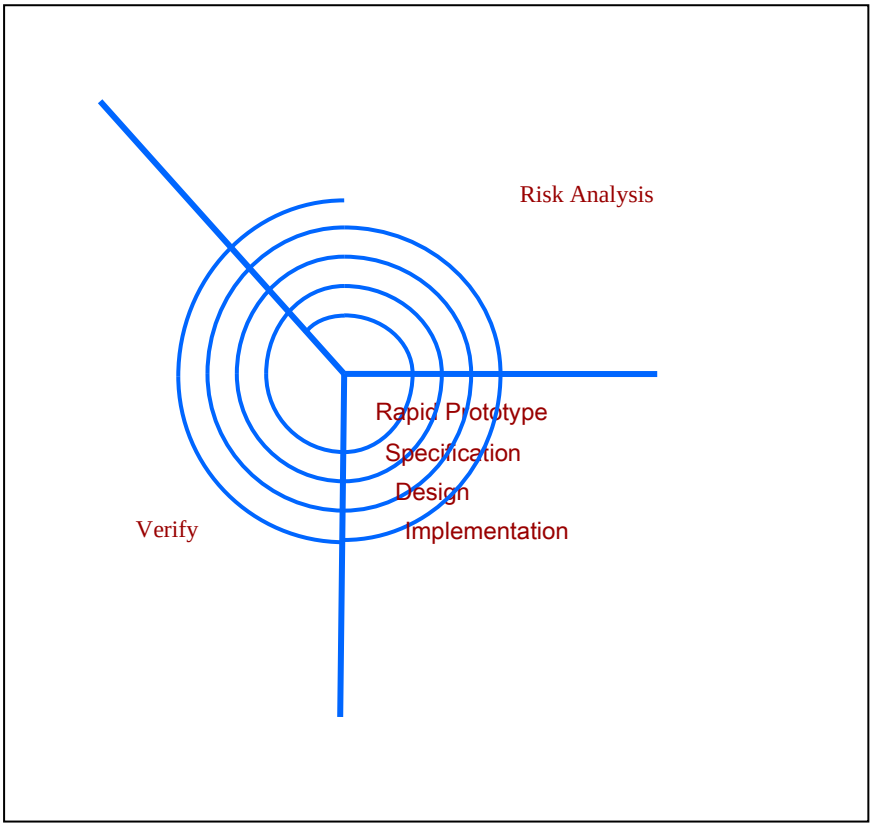
Synchronize and Stabilize Model

This is yet another form of incremental model adopted by Microsoft. In this model, during the requirements analysis interviews of potential customers are conducted and requirements document is developed. Once these requirements have been captured, specifications are drawn up. The project is then divided into 3 or 4 builds. Each build is carried out by small teams working in parallel. At the end of each day the code is synchronized (test and debug) and at the end of the build it is stabilized by freezing the build and removing any remaining defects. Because of the synchronizations, components always work together. The presence of an executable provides early insights into operation of product.

Spiral Model

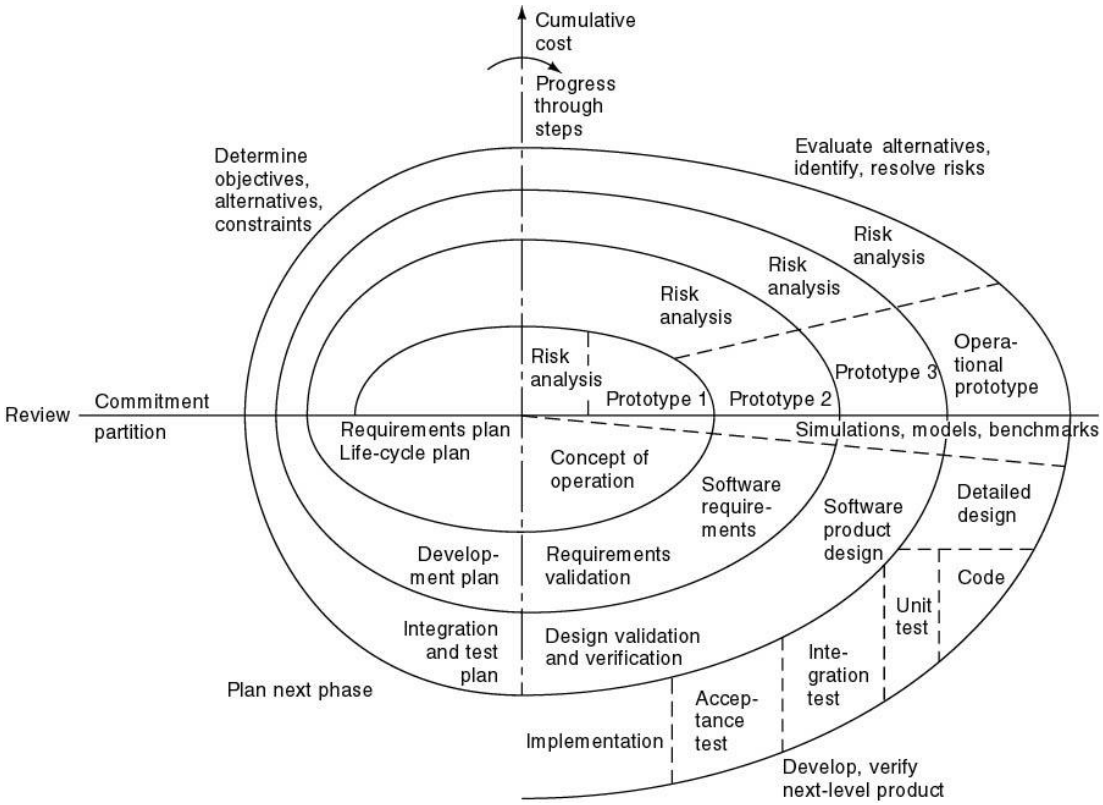
This model was developed by Barry Boehm. The main idea of this model is to avert risk as there is always an element of risk in development of software. For example, key personnel may resign at a critical juncture, the manufacturer of the software development may go bankrupt, etc.

In its simplified form, the Spiral Model is Waterfall model plus risk analysis. In this case each stage is preceded by identification of alternatives and risk analysis and is then followed by evaluation and planning for the next phase. If risks cannot be resolved, project is immediately terminated. This is depicted in the following diagram.

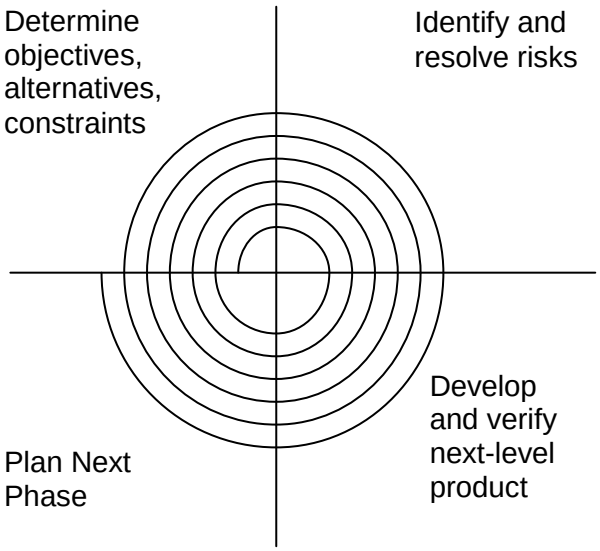


As can be seen, a Spiral Model has two dimensions. Radial dimension represents the cumulative cost to date and the angular dimension represents the progress through the spiral. Each phase begins by determining objectives of that phase and at each phase a new process model may be followed.

A full version of the Spiral Model is shown below:



The main strength of the Spiral Model comes from the fact that it is very sensitive to the risk. Because of the spiral nature of development it is easy to judge how much to test and there is no distinction between development and maintenance. It however can only be used for large-scale software development and that too for internal (in-house) software only.


Determine objectives, alternatives, constraints
Identify and resolve risks
Develop and verify next-level product
Plan Next Phase

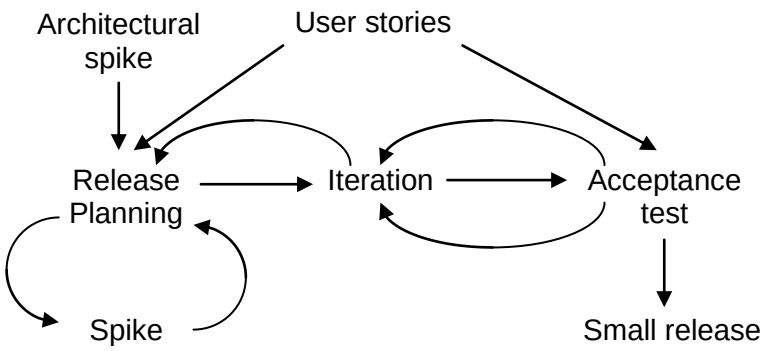
Lecture No. 5

Object-Oriented Lifecycle Models

Object-oriented lifecycle models appreciate the need for iteration within and between phases. There are a number of these models. All of these models incorporate some form of iteration, parallelism, and incremental development.

eXtreme Programming

It is a somewhat controversial new approach. In this approach user requirements are captured through stories which are the scenarios presenting the features needed by the client? Estimate for duration and cost of each story is then carried out. Stories for the next build are selected. Then each build is divided into tasks. Test cases for task are drawn up first before and development and continuous testing is performed throughout the development process.



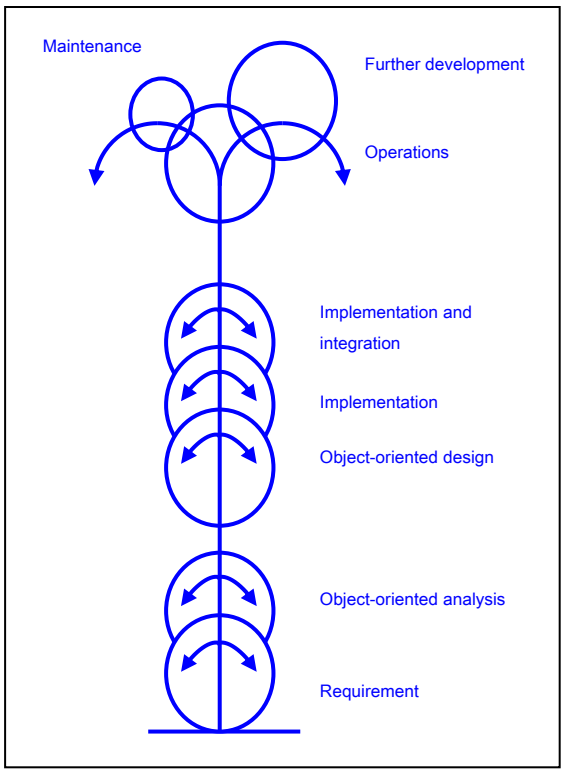
One very important feature of eXtreme programming is the concept of pair programming. In this, a team of two developers develop the software, working in team as a pair to the extent that they even share a single computer.

In eXtereme Programming model, computers are put in center of large room lined with cubicles and client representative is always present. One very important restriction imposed in the model is that no team is allowed to work overtime for 2 successive weeks.

XP has had some successes. It is good when requirements are vague or changing and the overall scope of the project is limited. It is however too soon to evaluate XP.

Fountain Model

Fountain model is another object-oriented lifecycle model. This is depicted in the following diagram.



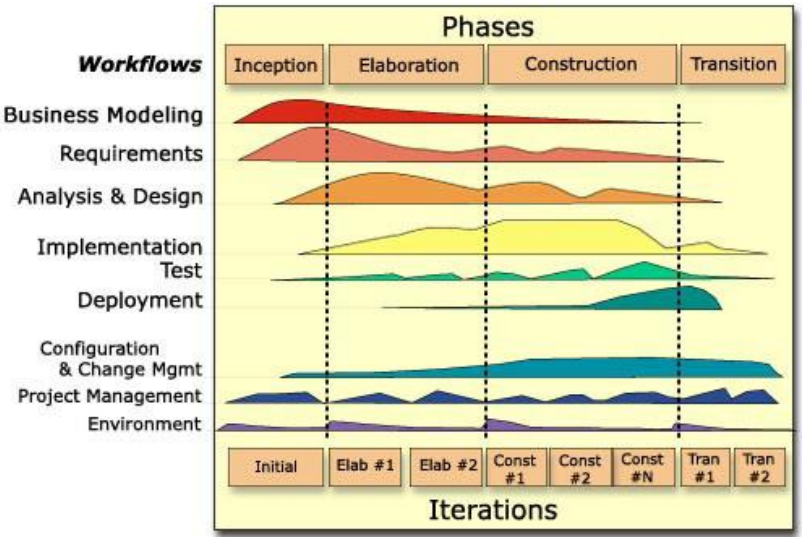
In this model the circles representing the various phases overlap, explicitly representing an overlap between activities. The arrows within a phase represent iteration within the phase. The maintenance cycle is smaller, to symbolize reduced maintenance effort when the object oriented paradigm is used.

Rational Unified Process (RUP)

Rational Unified Process is very closely associated with UML and Krutchen's architectural model.

In this model a software product is designed and built in a succession of incremental iterations. It incorporates early testing and validation of design ideas and early risk mitigation. The horizontal dimension represents the *dynamic aspect* of the process. This includes cycles, phases, iterations, and milestones. The vertical dimension represents the *static aspect* of the process described in terms of process components which include activities, disciplines, artifacts, and roles. The process emphasizes that during development, all activities are performed in parallel, however, and at a given time one activity may have more emphasis than the other.

The following figure depicting RUP is taken from Krutchen's paper.



Comparison of Lifecycle Models

As discussed above, each lifecycle model has some strengths and weaknesses. These are summarized in the following table:

Comparison of Process Models

Process Model	Strengths	Weaknesses
Build-and-fix model	• Fine for short programs that not require maintenance	• Totally unsatisfactory for nontrivial programs
Waterfall model	• Disciplined approach • Document driven	• Delivered product may not meet client's needs
Rapid prototyping model	• Ensures that delivered product meets client's needs	• Not yet proven
Incremental model	• Maximizes early return on investment • Promotes maintainability	• Requires open architecture • May degenerate into build-and-fix
Extreme programming	• Maximizes early return on investment • Fine when requirements are vague	• Small projects, small teams • Lack of design documentation • Has not yet been widely used
Synchronize-and-stabilize model	• Components always work together • Early insights into operation of product	• Has not been widely used other than at Microsoft
Spiral model	• "How much to test ?" in terms of risk • Maintenance is another cycle	• Only for large-scale, in-house products
Object-oriented models	• Iteration within phases • Parallelism between phases	• May degenerate into CABTAB

The criteria to be used for deciding on a model include the organization, its management, skills of the employees, and the nature of the product. No single model may fulfill the needs in a given situation. It may therefore be best to devise a lifecycle model tuned to your own needs by creating a "Mix-and-match" life-cycle model.

Quality Assurance and Documentation

It may be noted that there is no separate QA or documentation phase. QA is an activity performed throughout software production. It involves verification and validation.

Verification is performed at the end of each phase whereas validation is performed before delivering the product to the client.

Similarly, every phase must be fully documented before starting the next phase. It is important to note that postponed documentation may never be completed as the responsible individual may leave. Documentation is important as the product is constantly changing—we need the documentation to do this. The design (for example) will be modified during development, but the original designers may not be available to document it.

The following table shows the QA and documentation activities associated with each stage.

Phase	Documents	QA
Requirement Definition	• Rapid prototype, or • Requirements document	• Rapid prototype • Reviews
Functional Specification	• Specification document (specifications) • Software Product Management Plan	• Traceability • FS Review • Check the SPMP
Design	• Architectural Design • Detailed Design	• Traceability • Review
Coding	• Source code • Test cases	• Traceability • Review • Testing
Integration	• Source code • Test cases	• Integration testing • Acceptance testing
Maintenance	• Change record • Regression test cases	• Regression testing

Lecture No. 6

Software Project Management Concepts

Software project management is a very important activity for successful projects. In fact, in an organization at CMM Level basic project management processes are established to track cost, schedule, and functionality. That is, it is characterized by basic project management practices. It also implies that without project management not much can be achieved. Capers Jones, in his book on Software Best Practices, notes that, for the projects they have analyzed, good project management was associated with 100% of the successful project and bad project management was associated with 100% of the unsuccessful projects. Therefore, understanding of good project management principles and practices is essential for all project managers and software engineers.

Software project management involves that planning, organization, monitoring, and control of the people and the processes.

Software Project Management: Factors that influence results

The first step towards better project management is the comprehension of the factors that influence results of a project. Among these, the most important factors are:

- Project size

As the project size increases, the complexity of the problem also increases and therefore its management also becomes more difficult.

- Delivery deadline

Delivery deadline directly influences the resources and quality. With a realistic deadline, chances of delivering the product with high quality and reasonable resources increase tremendously as compared to an unrealistic deadline. So a project manager has to first determine a realistic and reasonable deadline and then monitor the project progress and ensure timely delivery.

- Budgets and costs

A project manager is responsible for ensuring delivery of the project within the allocated budget and schedule. A good estimate of budget, cost and schedule is essential for any successful project. It is therefore imperative that the project manager understand and learns the techniques and principle needed to develop these estimates.

- Application domain

Application domain also plays an important role in the success of a project. The chances of success of a project in a well-known application domain would be much better than of a project in a relatively unknown domain. The project manager thus needs to implement measures to handle unforeseen problems that may arise during the project lifecycle.

- Technology to be implemented

Technology also plays a very significant role in the success or failure of a project. One the one hand, a new “state-of-the-art” technology may increase the productivity of the team and quality of the product. On the other hand, it may prove to be unstable and hence

prove to be difficult to handle. Resultantly, it may totally blow you off the track. So, the project manager should be careful in choosing the implementation technology and must take proper safeguard measures.

- System constraints

The non-functional requirement or system constraints specify the conditions and the restrictions imposed on the system. A system that fulfils all its functional requirements but does not satisfy the non-functional requirements would be rejected by the user.

- User requirements

A system has to satisfy its user requirements. Failing to do so would render this system unusable.

- Available resources

A project has to be developed using the available resources who know the domain as well as the technology. The project manager has to ensure that the required number of resources with appropriate skill-set is available to the project.

Project Management Concerns

In order to plan and run a project successfully, a project manager needs to worry about the following issues:

1. Product quality: what would be the acceptable quality level for this particular project and how could it be ensured?
2. Risk assessment: what would be the potential problems that could jeopardize the project and how could they be mitigated?
3. Measurement: how could the size, productivity, quality and other important factors be measured and benchmarked?
4. Cost estimation: how could cost of the project be estimated?
5. Project schedule: how could the schedule for the project be computed and estimated?
6. Customer communication: what kind of communication with the customer would be needed and how could it be established and maintained consistently?
7. Staffing: how many people with what kind of resources would be needed and how that requirement could be fulfilled?
8. Other resources: what other hardware and software resources would be needed for the project?
9. Project monitoring: how the progress of the project could be monitored?

Thorough understanding and appreciation of these issues leads to the quest for finding satisfactory answers to these problems and improves the chances for success of a project.

Why Projects Fail?

A project manager is tasked to ensure the successful development of a product. Success cannot be attained without understanding the reasons for failure. The main reasons for the failure of software projects are:

1. changing customer requirements
2. ambiguous/incomplete requirements

3. unrealistic deadline
4. an honest underestimate of effort
5. predictable and/or unpredictable risks
6. technical difficulties
7. miscommunication among project staff
8. failure in project management

The first two points relate to good requirement engineering practices. Unstable user requirements and continuous requirement creep has been identified as the top most reason for project failure. Ambiguous and incomplete requirements lead to undesirable product that is rejected by the user.

As discussed earlier, delivery deadline directly influences the resources and quality. With a realistic deadline, chances of delivering the product with high quality and reasonable resources increase tremendously as compared to an unrealistic deadline. An unrealistic deadline could be enforced by the management or the client or it could be due to error in estimation. In both these cases it often results in disaster for the project.

A project manager who is not prepared and without a contingency plan for all sorts of predictable and unpredictable risks would put the project in jeopardy if such a risk should happen. Risk assessment and anticipation of technical and other difficulties allows the project manager to cope with these situations.

Miscommunication among the project staff is another very important reason for project failure. Lack of proper coordination and communication in a project results in wastage of resources and chaos.

The Management Spectrum

Effective project management focuses on four aspects of the project known as the 4 P's. These are: people, product, process, and project.

People

Software development is a highly people intensive activity. In this business, the software factory comprises of the people working there. Hence taking care of the first P, that is people, should take the highest priority on a project manager's agenda.

Product

The product is the outcome of the project. It includes all kinds of the software systems. No meaningful planning for a project can be carried-out until all the dimensions of the product including its functional as well as non-functional requirements are understood and all technical and management constraints are identified.

Process

Once the product objectives and scope have been determined, a proper software development process and lifecycle model must be chosen to identify the required work products and define the milestones in order to ensure streamlined development activities. It includes the set of all the framework activities and software engineering tasks to get the job done.

Project

A project comprises of all work the required to make the product a reality. In order to avoid failure, a project manager and software engineer is required to build the software product in a controlled and organized fashion and run it like other projects found in more concrete domains.

We now discuss these 4 in more detail.

People

In a study published by IEEE, the project team was identified by the senior executives as the most important contributor to a successful software project. However, unfortunately, people are often taken for granted and do no get the attention and focus they deserve. There are a number of players that participate in software process and influence the outcome of the project. These include senior managers, project (technical) managers, practitioners, customers, and end-users. Senior managers define the business vision whereas the project managers plan, motivate, organize and control the practitioners who work to develop the software product. To be effective, the project team must be organized to use each individual to the best of his/her abilities. This job is carried out by the team leader.

Team Leader

Project management is a people intensive activity. It needs the right mix of people skills. Therefore, competent practitioners often make poor team leaders.

Leaders should apply a problem solving management style. That is, a project manager should concentrate on understanding the problem to be solved, managing the flow of ideas, and at the same time, letting everyone on the team know that quality counts and that it will not be compromised.
MOI model of leadership developed by Weinberg suggest that a leadership needs Motivation, Organization, and Innovation.

Motivation is the ability to encourage technical people to produce to their best. **Organization** is the ability to mold the existing processes (or invent new ones) that will enable the initial concept to be translated into a final product, and **Idea or Innovation** is the ability to encourage people to create and feel creative.

It is suggested that successful project managers apply a problem solving management style. This involves developing an understanding of the problem and motivating the team to generate ideas to solve the problem.

Edgemon suggests that the following characteristics are needed to become an effective project manager:

- Problem Solving
  - Should be able to diagnose technical and organizational issues and be willing to change direction if needed.
- Managerial Identity
  - Must have the confidence to take control when necessary

- Achievement
  - Reward initiative (controlled risk taking) and accomplishment
- Influence and team building
  - Must remain under control in high stress conditions. Should be able to read signals and address peoples' needs.

DeMarco says that a good leader possesses the following four characteristics:

- Heart: the leader should have a big heart.
- Nose: the leader should have good nose to spot the trouble and bad smell in the project.
- Gut: the leader should have the ability to make quick decisions on gut feeling.
- Soul: the leader should be the soul of the team.

If analyzed closely, all these researchers seem to say essentially the same thing and they actually complement each other's point of view.

Lecture No. 7

The Software Team

There are many possible organizational structures. In order to identify the most suitable structure, the following factors must be considered:

- the difficulty of the problem to be solved
- the size of the resultant program(s) in lines of code or function points
- the time that the team will stay together (team lifetime)
- the degree to which the problem can be modularized
- the required quality and reliability of the system to be built
- the rigidity of the delivery date
- the degree of sociability (communication) required for the project

Constantine suggests that teams could be organized in the following generic structural paradigms:

- **closed paradigm**—structures a team along a traditional hierarchy of authority
- **random paradigm**—structures a team loosely and depends on individual initiative of the team members
- **open paradigm**—attempts to structure a team in a manner that achieves some of the controls associated with the closed paradigm but also much of the innovation that occurs when using the random paradigm
- **synchronous paradigm**—relies on the natural compartmentalization of a problem and organizes team members to work on pieces of the problem with little active communication among themselves

Mantei suggests the following three generic team organizations:

- Democratic decentralized (DD)

In this organization there is no permanent leader and task coordinators are appointed for short duration. Decisions on problems and approach are made by group consensus and communication among team is horizontal.

- Controlled decentralized (CD)

In CD, there is a defined leader who coordinates specific tasks. However, problem solving remains a group activity and communication among subgroups and individuals is horizontal. Vertical communication along the control hierarchy also occurs.

- Controlled centralized (CC)

In a Controlled Centralized structure, top level problem solving and internal team coordination are managed by the team leader and communication between the leader and team members is vertical.

Centralized structures complete tasks faster and are most useful for handling simple problems. On the other hand, decentralized teams generate more and better solutions than individuals and are most useful for complex problems

For the team morale point of view, DD is better.

Coordination and Communication Issues

Lack of coordination results in confusion and uncertainty. On the other hand, performance is inversely proportional to the amount of communication and hence too much communication and coordination is also not healthy for the project. Very large projects are best addressed with CC or CD when sub-grouping can be easily accommodated.

Kraul and Steeter categorize the project coordination techniques as follows:

- Formal, impersonal approaches

In these approaches, coordination is achieved through impersonal and formal mechanism such as SE documents, technical memos, schedules, error tracking reports.

- Formal, interpersonal procedures

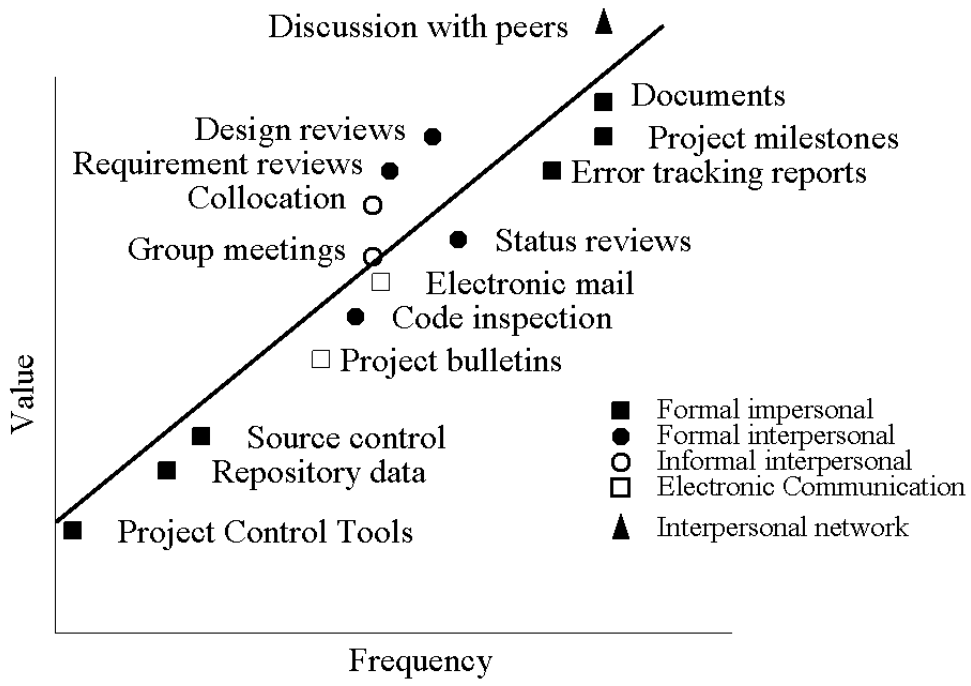
In this case, the approaches are interpersonal and formal. These include QA activities, design and code reviews, and status meetings.

- Informal, interpersonal procedures

This approach employs informal interpersonal procedures and includes group meetings and collocating different groups together.

- Electronic communication includes emails and bulletin boards.
- Interpersonal networking includes informal discussions with group members

The effectiveness of these approaches has been summarized in the following diagram:



Techniques that fall above the regression line yield more value to use ratio as compared to the ones below the line.

The Product: Defining the Problem

In order to develop an estimate and plan for the project, the scope of the problem must be established. This includes context, information objectives, and function and performance requirements. The estimate and plan is then developed by decomposing the problem and establishing a functional partitioning.

The Process

The next step is to decide which process model to pick. The project manager has to look at the characteristics of the product to be built and the project environment. For examples, for a relatively small project that is similar to past efforts, degree of uncertainty is minimized and hence Waterfall or linear sequential model could be used. For tight timelines, heavily compartmentalized, and known domain, RAD model would be more suitable. Projects with large functionality, quick turn around time are best developed incrementally and for a project in which requirements are uncertain, prototyping model will be more suitable.

Lecture No. 8

The Project Management

As discussed earlier, a project manager must understand what can go wrong and how to do it right. Reel has defined a 5 step process to improve the chances of success. These are:

- Start on the right foot: this is accomplished by putting in the required effort to understand the problem, set realistic objectives, build the right team, and provide the needed infrastructure.
- Maintain momentum: many projects, after starting on the right, loose focus and momentum. The initial momentum must be maintained till the very end.
- Track progress: no planning is useful if the progress is not tracked. Tracking ensures timely delivery and remedial action, if needed, in a suitable manner.
- Make smart decisions
- Conduct a postmortem analysis: in order to learn from the mistakes and improve the process continuously, a project postmortem must be conducted.

W5HH Principle

Barry Boehm has suggested a systematic approach to project management. It is known as the WWWWWHH principle. It comprises of 7 questions. Finding the answers to these 7 questions is essentially all a project manager has to do. These are:

- **WHY** is the system being developed?
- **WHAT** will be done?
- By **WHEN**?
- **WHO** is responsible for a function?
- **WHERE** they are organizationally located?
- **HOW** will the job be done technically and managerially?
- **HOW MUCH** of each resource (e.g., people, software, tools, database) will be needed?

Boehm's W^5HH principle is applicable, regardless of the size and complexity of the project and provide excellent planning outline.

Critical Practices

The Airlie Council has developed a list of critical success practices that must be present for successful project management. These are:

- Formal risk analysis
- Empirical cost and schedule estimation
- Metrics-based project management
- Earned value tracking
- Defect tracking against quality targets
- People aware project management

Finding the solution to these practices is the key to successful projects. We'll therefore spend a considerable amount of time in elaborating these practices.

© Virtual University of Pakistan
thegradup.com

Lecture No. 9

Software Size Estimation

The size of the software needs to be estimated to figure out the time needed in terms of calendar and man months as well as the number and type of resources required carrying out the job. The time and resources estimation eventually plays a significant role in determining the cost of the project.

Most organizations use their previous experience to estimate the size and hence the resource and time requirements for the project. If not quantified, this estimate is subjective and is as good as the person who is conducting this exercise. At times this makes it highly contentious. It is therefore imperative for a government organization to adopt an estimation mechanism that is:

1. Objective in nature.
2. It should be an acceptable standard with wide spread use and acceptance level.
3. It should serve as a single yardstick to measure and make comparisons.
4. Must be based upon a deliverable that is meaningful to the intended audience.
5. It should be independent of the tool and technology used for the developing the software.

A number of techniques and tools can be used in estimating the size of the software. These include:

1. Lines of code (LOC)
2. Number of objects
3. Number of GUIs
4. Number of document pages
5. Functional points (FP)

Comparison of LOC and FPA

Out of these 5, the two most widely used metrics for the measurement of software size are FP and LOC. LOC metric suffer from the following shortcomings:

1. There are a number of questions regarding the definition for lines of code. These include:
    a. Whether to count physical line or logical lines?
    b. What type of lines should be counted? For example, should the comments, data definitions, and blank lines be counted or not?
2. LOC is heavily dependent upon the individual programming style.
3. It is dependent upon the technology and hence it is difficult to compare applications developed in two different languages. This is true for even seemingly very close languages like in C++ and Java.
4. If a mixture of languages and tools is used then the comparison is even more difficult. For example, it is not possible to compare a project that delivers a 100,000-line mixture of Assembly, C++, SQL and Visual Basic to one that delivers 100,000 lines of COBOL.

FP measures the size of the functionality provided by the software. The functionally is measured as a function of the data and the operations performed on that data. The measure is independent of the tool and technology used and hence provides a consistent measure for comparison between various organizations and projects.

The biggest advantage of FP over LOC is that LOC can be counted only AFTER the code has been developed while FP can be counted even at the requirement phase and hence can be used for planning and estimation while the LOC cannot be used for this purpose.

Another major distinction between the FP and LOC is that the LOC measures the application from a developer's perspective while the FP is a measure of the size of the functionality from the user's perspective. The user's view, as defined by IFPUG, is as follows:

> A *user view* is a description of the business functions and is approved by the user. It represents a formal description of the user's business needs in the user's language. It can vary in physical form (e.g., catalog of transactions, proposals, requirements document, external specifications, detailed specifications, user handbook). Developers translate the user information into information technology language in order to provide a solution. Function point counts the application size from the user's point of view. It is accomplished using the information in a language that is common to both user(s) and developers.

Therefore, Function Point Analysis measures the size of the functionality delivered and used by the end user as opposed to the volume of the artifacts and code.

	Assembler Version	Ada Version	Difference
Source Code Size	100,000	25,000	-75,000
Activity - in person months			
Requirements	10	10	0
Design	25	25	0
Coding	100	20	-80
Documentation	15	15	0
Integration and Testing	25	15	-10
Management	25	15	-10
Total Effort	200	100	-100
Total Cost	$1,000,000	$500,000	-$500,000
Cost Per Line	$10	$20	$10
Lines Per Person-Month	500	250	-250

The Paradox of Reversed Productivity for High-Level Languages
Consider the following example:

In this example, it is assumed that the same functionality is implemented in Assembly and Ada. As coding in Assembly is much more difficult and time consuming as compared to Ada, it takes more time and it is also lengthy. Because there is a huge difference in the code size in terms of Lines of Code, the cost per line in case of Assembly is much less as compared to Ada. Hence coding in Assembly appears to be more cost effective than Ada while in reality it is not. This is a paradox!

Function Point Analysis - A Brief History and Usage

In the mid 70's, IBM felt the need to establish a more effective and better measure of system size to predict the delivery of software. It commissioned Allan Albrecht to lead this effort. As a result he developed this approach which today known as the Function Point Analysis. After several years of internal use, Albrecht introduced the methodology at a joint/share conference. From 1979 to 1984 continued statistical analysis was performed on the method and refinements were made. At that point, a non-profit organization by the name of International Function Point User Group (IFPUG) was formed which formally took onto itself the role of refining and defining the counting rules. The result is the function point methodology that we use today.

Since 1979, when Albrecht published his first paper on FP, its popularity and use has been increasing consistently and today it is being used as a de facto standard for software measurement. Following is a short list of organizations using FP for estimation:

1. IEEE recommends it for use in productivity measurement and reporting.
2. Several governments including UK, Canada, and Hong Kong have been using it and it has been recommended to these governments that all public sector project use FP as a standard for the measurement of the software size.
3. Government of the Australian state Victoria has been using FP since 1997 for managing and outsourcing projects to the tune of US$ 50 Million every year.
4. In the US several large government departments including IRS have adopted FP analysis as a standard for outsourcing, measurement, and control of software projects.
5. A number of big organizations including Digital Corporation and IBM have been using FP for their internal use for the last many years.

Usage of FP includes:

- Effort Scope Estimation
- Project Planning
- Determine the impact of additional or changed requirements
- Resource Planning/Allocation
- Benchmarking and target setting
- Contract Negotiations

Following is a list of some of the FP based metrics used for these purposes:

- Size – Function Points
- Defects – Per Function Point
- Effort – Staff-Months
- Productivity – Function Points per Staff-Month
- Duration – Schedule (Calendar) Months

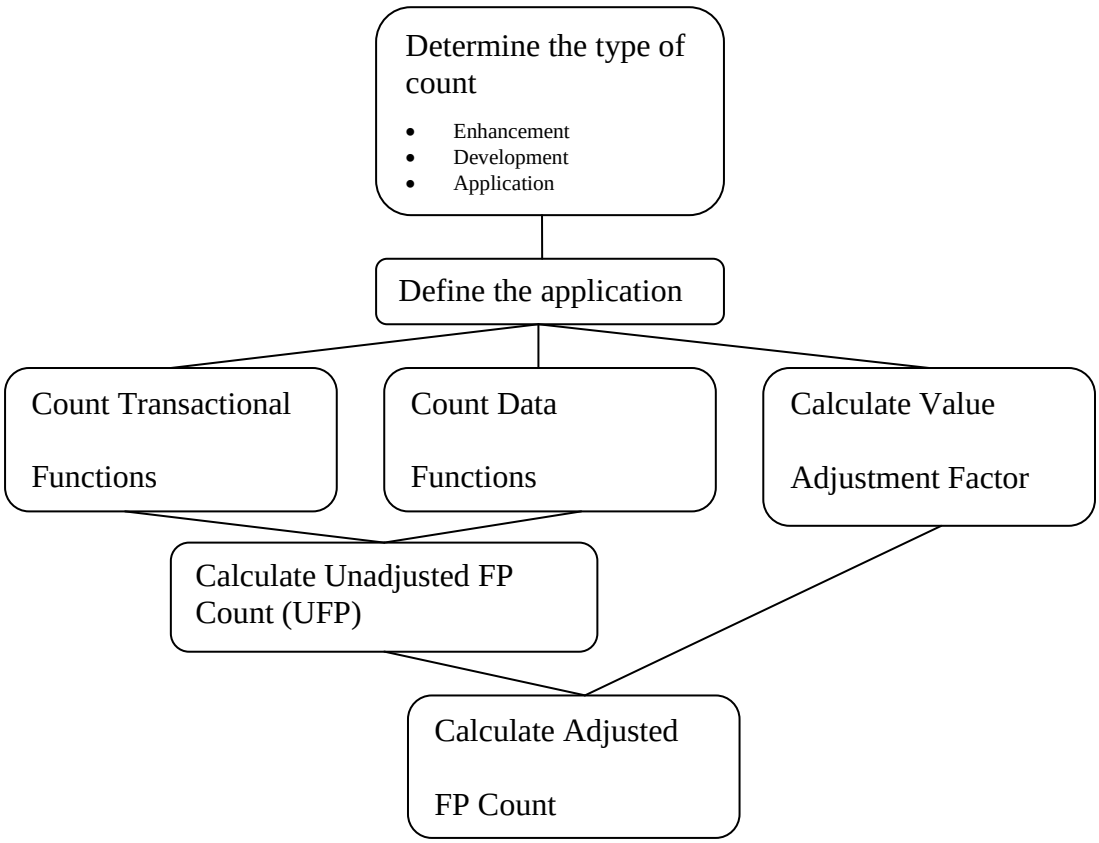
- Time Efficiency – Function Points per Month
- Cost – Per Function

© Virtual University of Pakistan
thegradup.com

Lecture No. 10

Function Point Counting Process

The following diagram depicts the function point counting process.



These steps are elaborated in the following subsections. The terms and definitions are the ones used by IFPUG and have been taken directly from the IFPUG Function Point Counting Practices Manual (CPM) Release 4.1. The following can therefore be treated as an abridged version of the IFPUG CPM Release 4.1.

Determining the type of count

A Function Point count may be divided into the following types:

1. **Development Count**: A development function point count includes all functions impacted (built or customized) by the project activities.
2. **Enhancement Count**: An enhancement function point count includes all the functions being added, changed and deleted. The boundary of the application(s) impacted remains the same. The functionality of the application(s) reflects the impact of the functions being added, changed or deleted.
3. **Application Count**: An application function point count may include, depending on the purpose (e.g., provide a package as the software solution):
   a) only the functions being used by the user

b) all the functions delivered
c) The application boundary of the two counts is the same and is independent of the *scop*e.

Defining the Application Boundary

The application boundary is basically the system context diagram and determines the scope of the count. It indicates the border between the software being measured and the user. It is the conceptual interface between the 'internal' application and the 'external' user world. It depends upon the user's external view of the system and is independent of the tool and technology used to accomplish the task.

The position of the application boundary is important because it impacts the result of the function point count. The application boundary assists in identifying the data entering the application that will be included in the scope of the count.

Count Data Functions

Count of the data functions is contribution of the data manipulated and used by the application towards the final function point count. The data is divided into two categories: the *Internal Logical Files* (ILF) and the *External Interface Files* (EIF). These and the related concepts are defined and explained as follows.

Internal Logical Files (ILF)

An internal logical file (ILF) is a user identifiable group of logically related data or control information maintained within the boundary of the application. The primary intent of an ILF is to hold data maintained through one or more elementary processes of the application being counted.

External Interface Files

An external interface file (EIF) is a user identifiable group of logically related data or control information referenced by the application, but maintained within the boundary of another application. The primary intent of an EIF is to hold data referenced through one or more elementary processes within the boundary of the application counted. This means an EIF counted for an application must be in an ILF in another application.

Difference between ILFs and EIFs

The primary difference between an internal logical file and an external interface file is that an EIF **is not** maintained by the application being counted, while an ILF is.

Definitions for Embedded Terms

The following paragraphs further define ILFs and EIFs by defining embedded terms within the definitions.

Control Information

Control Information is data that influences an elementary process of the application being counted. It specifies what, when, or how data is to be processed. For example, someone in the payroll department establishes payment cycles to schedule when the employees for each location are to be paid. The payment cycle, or schedule, contains timing information that affects when the elementary process of paying employees occurs.

User Identifiable

The term *user identifiable* refers to defined requirements for processes and/or groups of data that are agreed upon, and understood by, both the user(s) and software developer(s). For example, users and software developers agree that a Human Resources Application will maintain and store Employee information in the application.

Maintained

The term *maintained* is the ability to modify data through an elementary process. Examples include, but are not limited to, add, change, delete, populate, revise, update, assign, and create.

Elementary Process

An *elementary process* is the smallest unit of activity that is meaningful to the user(s). For example, a user requires the ability to add a new employee to the application. The user definition of employee includes salary and dependent information. From the user perspective, the smallest unit of activity is to add a new employee. Adding one of the pieces of information, such as salary or dependent, is not an activity that would qualify as an elementary process. The *elementary process* must be self-contained and leave the business of the application being counted in a consistent state. For example, the user requirements to add an employee include setting up salary and dependent information. If all the employee information is not added, an employee has not yet been created. Adding some of the information alone leaves the business of adding an employee in an inconsistent state. If both the employee salary and dependent information is added, this unit of activity is completed and the business is left in a consistent state.

ILF/EIF Counting Rules

This section defines the rules that apply when counting internal logical files and external interface files.

Summary of Counting Procedures

The ILF and EIF counting procedures include the following two activities:

1) Identify the ILFs and EIFs.
2) Determine the ILF or EIF complexity and their contribution to the unadjusted function point count. ILF and EIF counting rules are used for each activity.

There are two types of rules:

- Identification rules
- Complexity and contribution rules

The following list outlines how the rules are presented:
- ILF identification rules
- EIF identification rules
- Complexity and contribution rules, which include:
  - Data element types (DETs)
  - Record element types (RETs)

ILF Identification Rules

To identify ILFs, look for groups of data or control information that satisfy the definition of an ILF. All of the following counting rules must apply for the information to be counted as an ILF.

- The group of data or control information is logical and user identifiable.
- The group of data is maintained through an elementary process within the application boundary being counted.

EIF Identification Rules

To identify EIFs, look for groups of data or control information that satisfy the definition of an EIF. All of the following counting rules must apply for the information to be counted as an EIF.

- The group of data or control information is logical and user identifiable.
- The group of data is referenced by, and external to, the application being counted.
- The group of data **is not maintained** by the application being counted.
- The group of data is maintained in an ILF of another application.

Complexity and Contribution Definitions and Rules

The number of ILFs, EIFs, and their relative functional complexity determine the contribution of the data functions to the unadjusted function point count. Assign each identified ILF and EIF a functional complexity based on the number of data element types (DETs) and record element types (RETs) associated with the ILF or EIF. This section defines DETs and RETs and includes the counting rules for each.

DET Definition

A *data element type* is a unique user recognizable, non-repeated field.

DET Rules

The following rules apply when counting DETs:

1. Count a DET for each unique user recognizable, non-repeated field maintained in or retrieved from the ILF or EIF through the execution of an elementary process. For example:
   - An account number that is stored in multiple fields is counted as one DET.

- A before or after image for a group of 10 fields maintained for audit purposes would count as one DET for the before image (all 10 fields) and as one DET for the after image (all 10 fields) for a total of 2 DETs.
- The result(s) of a calculation from an elementary process, such as calculated sales tax value for a customer order maintained on an ILF is counted as one DET on the customer order ILF.
- Accessing the price of an item which is saved to a billing file or fields such as a time stamp if required by the user(s) are counted as DETs.
- If an employee number which appears twice in an ILF or EIF as (1) the key of the employee record and (2) a foreign key in the dependent record, count the DET only once.
- Within an ILF or EIF, count one DET for the 12 Monthly Budget Amount fields. Count one additional field to identify the applicable month. For Example:

2. When two applications maintain and/or reference the same ILF/EIF, but each maintains/references separate DETs, count only the DETs being used by each application to size the ILF/EIF. For Example:
   - Application A may specifically identify and use an address as street address, city, state and zip code. Application B may see the address as one block of data without regard to individual components. Application A would count four DETs; Application B would count one DET.
   - Application X maintains and/or references an ILF that contains a SSN, Name, Street Name, Mail Stop, City, State, and Zip. Application Z maintains and/or references the Name, City, and State. Application X would count seven DETs; Application Z would count three DETs.
3. Count a DET for each piece of data required by the user to establish a relationship with another ILF or EIF.
   - In an HR application, an employee's information is maintained on an ILF. The employee's job name is included as part of the employee's information. This DET is counted because it is required to relate an employee to a job that exists in the organization. This type of data element is referred to as a *foreign key*.
   - In an object oriented (OO) application, the user requires an association between object classes, which have been identified as separate ILFs. Location name is a DET in the Location EIF. The location name is required when processing employee information; consequently, it is also counted as a DET within the Employee ILF.

Lecture No. 11

Function Point Counting Process (cont.)

RET Definition

A *record element type* (RET) is a user recognizable subgroup of data elements within an ILF or EIF. There are two types of subgroups:

- Optional
- Mandatory

Optional subgroups are those that the user has the option of using one or none of the subgroups during an elementary process that adds or creates an instance of the data.
Mandatory subgroups are subgroups where the user must use at least one. For example, in a Human Resources Application, information for an employee is added by entering some general information. In addition to the general information, the employee is a salaried or hourly employee. The user has determined that an employee must be either salaried or hourly. Either type can have information about dependents. For this example, there are three subgroups or RETs as shown below:

- Salaried employee (mandatory); includes general information
- Hourly employee (mandatory); includes general information
- Dependent (optional)

RET Rules

One of the following rules applies when counting RETs:

- Count a RET for each optional or mandatory subgroup of the ILF or EIF.
  Or
- If there are no subgroups, count the ILF or EIF as one RET.

Hints to Help with Counting

The following hints may help you apply the ILF and EIF counting rules.
Caution: These hints *are not* rules and should not be used as rules.

1. Is the data a logical group that supports specific user requirements?
   a) An application can use an ILF or EIF in multiple processes, but the ILF or EIF is counted only once.
   b) A logical file cannot be counted as both an ILF and EIF for the same application. If the data group satisfies both rules, count as an ILF.
   c) If a group of data was not counted as an ILF or EIF itself, count its data elements as DETs for the ILF or EIF, which includes that group of data.
   d) Do not assume that one physical file, table or object class equals one logical file when viewing data logically from the user perspective.
   e) Although some storage technologies such as tables in a relational DBMS or sequential flat file or object classes relate closely to ILFs or EIFs, do not assume that this always equals a one-to-one physical-logical relationship.
   f) Do not assume all physical files must be counted or included as part of an ILF or EIF.
2. Where is data maintained? Inside or outside the application boundary?
   a) Look at the workflow.

b) In the process functional decomposition, identify where interfaces occur with the user and other applications.
c) Work through the process diagram to get hints.
d) Credit ILFs maintained by more than one application to each application at the time the application is counted. Only the DETs being used by each application being counted should be used to size the ILF/EIF.

3. Is the data in an ILF maintained through an elementary process of the application?
   a) An application can use an ILF or EIF multiple times, but you count the ILF or EIF only once.
   b) An elementary process can maintain more than one ILF.
   c) Work through the process diagram to get hints.
   d) Credit ILFs maintained by more than one application to each application at the time the application is counted.

Hints to Help with Identifying ILFs, EIFs, and RETs

Differentiating RETs from ILFs and EIFs is one of the most activities in FP analysis. Different concepts regarding entities play a pivotal role in this regards. Let us therefore understand what an entity is and what different types of entities are.

Entity

An entity is defined by different people as follows:

- A thing that can be distinctly identified. (Chen)
- Any distinguishable object that is to be represented in the database. (Date)
- Any distinguishable person, place, thing, event or concept about which information is kept. (Bruce)
- A data entity represents some "thing" that is to be stored for later reference. The term entity refers to the logical representation of data. (Finkelstein)
- An entity may also represent the relationship between two or more entities, called **associative entity**. (Reingruber)
- An entity may represent a subset of information relevant to an instance of an entity, called **subtype entity**. (Reingruber)

That is, an entity is a principal data object about which information is collected that is a fundamental thing of relevance to the user, about which a collection of facts is kept.

An entity can be a weak entity or a strong entity. A weak entity is the one which does not have any role in the problem domain without some other entity. Weak entities are RETs and strong entities are ILFs and EIFs. Identification of weak entities is therefore important for distinguishing between RETs and logical files.

Weak Entities

There are three types of weak entities: Associative entity types, attributive entity type, and entity subtype. These are elaborated as follows:

- **Associative Entity Type** – An entity which defines many-to-many relationship between two or more entities.
  - Student – course
  - Part – dealer

- **Attributive Entity Type –** An entity type which further describes one or more characteristics of another entity.
  - Product – Part
  - Product – Product Price Information
- **Entity Subtype** – A subdivision of entity. A subtype inherits all the attributes of its parent entity type, and may have additional, unique attributes.
  - **Employee**
    Permanent Employee
    Contract Employee

  - **Employee**
    Married Employee
    Single Employee

Logical Files

Grouping of data into logical files is the result of combined effect of two grouping methods:

- How data is accessed as a group by elementary processes? (process driven)
- The relationship between the entities and their interdependency based on business rules. (data driven)

The following guideline can be used for this purpose:

- Process Driven Approach
- Data Driven Approach

Process Driven Approach

If several entities are always **created** together and **deleted** together then this is a strong indication that they should be grouped into a single logical file.

- A customer PO is a single group of data from a user business perspective.
- It consists of a header and items information.
- From a business perspective, an order cannot be created unless it has at least one item and if the order is deleted both the order header and items are deleted. However the header and the items may have independent maintenance transactions.

Data Driven Approach

Entity Independence: an entity has significance to the business in and of itself without the presence of other entities. This is a logical file.

Entity Dependence: an entity is not meaningful, has no significance to the business in and of itself **without** the presence of other entities. This is an RET.

- Given two linked entities A and B, whether B is dependent or independent:
  - Is B significant to the business apart from the occurrence of A linked to it?
  - If we delete an occurrence "a" of A, what happens to occurrence "b" of B linked to "a"?

For example in the following scenarios, the first one is the example of entity dependence while the second one is the example of entity independence.

- Employee – Child
- Employee - Company Adopted Child

These concepts are summarized in the following table:

E/R Concept	E/R Term	FPA Term	IFPUG CPM 4.1 Definition
Principal data objects about which information is collected	Entity or Entity Type	ILF or EIF	File refers to a logically related group of data and not the physical implementation of those groups of data.
An entity type which contains attributes which further describe relationships between other entities	Associative entity type	Optional or mandatory subgroup	User recognizable subgroup of data elements within an ILF or EIF
An entity type that further describes one or more characteristics of another entity type	Attributive entity type	Optional or mandatory subgroup	User recognizable subgroup of data elements within an ILF or EIF
A division of an entity type, which inherits all the attributes and relationships of its parent entity type; may have additional, unique attributes and relationships	Entity subtype	Optional or mandatory subgroup	User recognizable subgroup of data elements within an ILF or EIF

Definitions: EIs, EOs and EQs

This section includes the definitions of EIs, EOs and EQs. Embedded terms within the definitions are defined, and examples are included throughout this definition section.

External Inputs

An external input (EI) is an elementary process that processes data or control information that comes from outside the application boundary. The primary intent of an EI is to maintain one or more ILFs and/or to alter the behavior of the system.

External Outputs

An external output (EO) is an elementary process that sends data or control information outside the application boundary. The primary intent of an external output is to present information to a user through processing logic other than, or in addition to, the retrieval of data or control information . The processing logic must contain at least one mathematical formula or calculation, or create derived data. An external output may also maintain one or more ILFs and/or alter the behavior of the system.

External Inquiry

An external inquiry (EQ) is an elementary process that sends data or control information outside the application boundary. The primary intent of an external inquiry is to present information to a user through the retrieval of data or control information from an ILF or EIF. The processing logic contains no mathematical formulas or calculations, and creates no derived data. No ILF is maintained during the processing, nor is the behavior of the system altered.

Summary of the Functions Performed by EIs, EOs, and EQs

The main difference between the transactional function types is their primary intent. The table below summarizes functions that may be performed by each transactional function type, and specifies the primary intent of each. Note the primary intent for an EI—this is

the main difference from EOs and EQs. Some of the differences between EOs and EQs are that an EO may perform the functions of altering the behavior of the system or maintaining one or more ILFs when performing the primary intent of presenting information to the user. Other differences are identified in the section below that summarizes forms of processing logic used by each transactional function.

	Transactional Function Type		
Function	EI	EO	EQ
Alter the behavior of the system	PI	F	N/A
Maintain one or more ILFs	PI	F	N/A
Present information to a user	F	PI	PI

Legend:

PI The primary intent of the transactional function type

F A function of the transactional function type, but is not the primary intent and is sometimes present

N/A The function is not allowed by the transactional function type.

Processing Logic

Processing logic is defined as requirements specifically requested by the user to complete an elementary process. Those requirements may include the following actions:

1. Validations are performed.

For example, when adding a new employee to an organization, the employee process has processing logic that validates the information being added.

2. Mathematical formulas and calculations are performed.

For example, when reporting on all employees within an organization the process includes calculating the total number of salaried employees, hourly employees and all employees.

3. Equivalent values are converted

For example, an elementary process references currency conversion rates from US dollars to other currencies. The conversion is accomplished by retrieving values from tables, so calculations need not be performed.

4. Data is filtered and selected by using specified criteria to compare multiple sets of data.

For example, to generate a list of employees by assignment, an elementary process compares the job number of a job assignment to select and lists the appropriate employees with that assignment.

5. Conditions are analyzed to determine which are applicable.

For example, processing logic exercised by the elementary process when an employee is added and will depend on whether an employee is paid based on salary or hours worked.

6. One or more ILFs are updated.

For example, when adding an employee, the elementary process updates the employee ILF to maintain the employee data.

7. One or more ILFs or EIFs are referenced.

For example, when adding an employee, the currency EIF is referenced to use the correct US dollar conversion rate to determine an employee's hourly rate.

8. Data or control information is retrieved.

a) For example, to view a list of possible pay grades, pay grade information is retrieved.

9. Derived data is created by transforming existing data to create additional data.

For example, to determine (derive) a patient's registration number (e.g., SMIJO01), the following data is concatenated:

a) the first three letters of the patient's last name (e.g., SMI for Smith)
b) the first two letter of the patient's first name (e.g., JO for John)
c) a unique two-digit sequence number (starting with 01)

10. Behavior of the system is altered.

For example, the behavior of the elementary process of paying employees is altered when a change is made to pay them every other Friday versus on the 15th and the last day of the month.

11. Prepare and present information outside the boundary.

For example, a list of employees displayed for the user.

12. Capability exists to accept data or control information that enters the application boundary.

For example, a user enters several pieces of information to add a customer order to the system.

13. Data is resorted or rearranged.

For example, a user requests the list of employees in alphabetical order.

Note: Resorting or rearranging a set of data does not impact the identification of the type or uniqueness of a transactional function.

Summary of Processing Logic Used by EIs, EOs and EQs

The following table summarizes which forms of g logic may be performed by EIs, Eos, and EQs. Foe each transactional function type, certain types of processing logic must be performed to accomplish the primary intent of that type.

Form of Processing Logic	Transactional Functional Type		
	EI	**EO**	**EQ**
1. Validations are performed	c	c	c
2. Mathematical Formula and calculations are performed	c	m*	n
3. Equivalent Values are converted	c	c	c
4. Data is filtered and selected by using specified criteria to compare multiple sets of data.	c	c	c
5. Conditions are analyzed to determine which are applicable	c	c	c
6. At least one ILF is updated	m*	m*	n
7. At least one ILF or EIF is referenced	c	c	m
8. Data or control information is retrieved	c	c	m
9. Derived data is created	c	m*	n
10. Behavior of system is altered	m*	m*	n
11. Prepare and present information outside the boundary	c	m	m
12. Capability to accept data or control information that enters the application boundary	m*	c	c
13. Resorting or rearranging a set of data	c	c	c

Legend

m	it is **mandatory** that the function type perform the form of processing logic.
m*	it is **mandatory** that the function type perform at least on of these (m*) forms of processing logic
c	the function type **can** perform the form of processing logic, but it is not mandatory.
n	function type **cannot** perform the form of processing logic

EI/EO/EQ Counting Rules

This section defines the rules that apply when counting EIs, EOs and EQs.

Elementary Process Identification Rules

To identify elementary processes, look for user activities occurring in the application.
All of the following counting rules must apply for the process to be identified as an elementary process.

- The process is the smallest unit of activity that is meaningful to the user.
- The process is self-contained and leaves the business of the application in a consistent state.

Transactional Functions Counting Rules

To classify each elementary process, determine which of the primary intent descriptions apply, and use the associated rules to identify a specific transactional function type.

Primary Intent Description for EIs

The primary intent of an elementary process is to maintain an ILF or alter the behavior of the system.

External Input Counting Rules

For each elementary process that has a primary intent to maintain one or more ILFs or to alter the behavior of the system, apply the following rules to determine if the function should be classified as an external input. All of the rules must apply for the elementary process to be counted as a unique occurrence of an external input.

- The data or control information is received from outside the application boundary.
- At least one ILF is maintained if the data entering the boundary is not control information that alters the behavior of the system.
- For the identified process, one of the following three statements must apply:
    - Processing logic is unique from the processing logic performed by other external inputs for the application.
    - The set of data elements identified is different from the sets identified for other external inputs for the application.
    - The ILFs or EIFs referenced are different from the files referenced by other external inputs in the application.

Primary Intent Description for EOs and Eqs

The primary intent of the elementary process is to present information to a user.

Shared EO and EQ Counting Rules

For each elementary process that has a primary intent to present information to a user, apply the following rules to determine if the process may be classified as an external output or external inquiry. All of the rules must apply for the elementary process to be counted as a unique occurrence of an external output or external inquiry.

- The function sends data or control information external to the application boundary.
- For the identified process, one of the following three statements must apply:
    - Processing logic is unique from the processing logic performed by other external outputs or external inquiries for the application.
    - The set of data elements identified is different from the sets identified for other external outputs and external inquiries in the application.
    - The ILFs or EIFs referenced are different from the files referenced by other external outputs and external inquiries in the application.

Additional External Output Counting Rules

In addition to adhering to all shared EO and EQ rules, one of the following rules must apply for the elementary process to be counted as a unique external output.

- The processing logic of the elementary process contains at least one mathematical formula or calculation.
- The processing logic of the elementary process creates derived data.
- The processing logic of the elementary process maintains at least one ILF.
- The processing logic of the elementary process alters the behavior of the system.

Additional External Inquiry Counting Rules

In addition to adhering to all shared EO and EQ rules, all of the following rules must apply for the elementary process to be counted as a unique external inquiry.

- The processing logic of the elementary process retrieves data or control information from an ILF or EIF.
- The processing logic of the elementary process does not contain a mathematical formula or calculation.
- The processing logic of the elementary process does not create derived data.
- The processing logic of the elementary process does not maintain an ILF.
- The processing logic of the elementary process does not alter the behavior of the system.

Complexity and Contribution Definitions and Rules

The number of EIs, EOs, and EQs and their relative functional complexities determine the contribution of the transactional functions to the unadjusted function point count.
Assign each identified EI, EO and EQ a functional complexity based on the number of file types referenced (FTRs) and data element types (DETs).

FTR Definition

A *file type referenced* is
- An internal logical file read or maintained by a transactional function or
- An external interface file read by a transactional function

DET Definition

A *data element type* is a unique user recognizable, non-repeated field.

EI Complexity and Contribution Rules

This section defines FTR and DET rules used to determine the complexity and contribution of external inputs.

FTR Rules for an EI

The following rules apply when counting FTRs:

- Count an FTR for each ILF maintained.
- Count an FTR for each ILF or EIF read during the processing of the external input.
- Count only one FTR for each ILF that is both maintained and read.

DET Rules for an EI

The following rules apply when counting DETs:

- Count one DET for each user recognizable, non-repeated field that enters or exits the application boundary and is required to complete the external input. For example, job name and pay grade are two fields that the user provides when adding a job.
- Do not count fields that are retrieved or derived by the system and stored on an ILF during the elementary process if the fields did not cross the application boundary.

For example, when the customer order is added to the system, the unit price is automatically retrieved for each ordered item and stored on the billing record. The unit price would not be counted as a DET for the EI because it did not cross the boundary when the user adds the customer order.

For example, in order to maintain the US hourly rate for hourly employees working in other countries with other currencies, the local hourly rate is provided by the user. During the processing of all the pieces of data provided to add an employee, a conversion rate is retrieved from the currency system to calculate the US hourly rate. The calculated US hourly rate is maintained on the employee ILF as a result of adding the employee. The US hourly rate would not be counted as a DET for the EI because it does not enter the boundary, but is internally calculated (i.e., it is derived data).

- Count one DET for the capability to send a system response message outside the application boundary to indicate an error occurred during processing, confirm that processing is complete or verify that processing should continue.

For example, if a user tries to add an existing employee to a Human Resources application, the system generates one of several error messages and the incorrect field is highlighted. Count one DET that includes all the system responses which indicate the error conditions, confirm that processing is complete or verify that processing should continue.

- Count one DET for the ability to specify an action to be taken even if there are multiple methods for invoking the same logical process.

For example, if the user can initiate the adding of an employee clicking on the OK button or by pressing a PF key, count one DET for the ability to initiate the process.

EO/EQ Complexity and Contribution Rules

This section defines FTR and DET rules used to determine the complexity and contribution of external outputs and external inquiries.

Shared FTR Rules for EOs and EQs

The following rule applies when counting FTRs for both EOs and EQs:

- Count one FTR for each ILF or EIF read during the processing of the elementary process.

Additional FTR Rules for an EO

The following additional rules apply when counting FTRs for EOs:

- Count one FTR for each ILF maintained during the processing of the elementary process.
- Count only one FTR for each ILF that is both maintained and read during the elementary process.

Shared DET Rules for EOs and EQs

The following rules apply when counting DETs for both EOs and EQs.

- Count one DET for each user recognizable, non-repeated field that enters the application boundary and is required to specify when, what and/or how the data is to be retrieved or generated by the elementary process. For example (EO/EQ), to generate a list of employees, employee name is a field the user provides when indicating which employees to list.
- Count one DET for each user recognizable, non-repeated field that exits the boundary.

For example (EO/EQ), a text message may be a single word, sentence, or phrase—a line or paragraph included on a report to indicate an explanatory comment counts as a single DET.

For example (EO/EQ), an account number or date physically stored in multiple fields is counted as one DET when it is required as a single piece of information.

For example (EO/EQ), a pie chart might have a category label and a numerical equivalent in a graphical output. Count two DETs —one for designating the category and one for the numerical value.

- If a DET both enters and exits the boundary, count it only once for the elementary process.
- Count one DET for the capability to send a system response message outside the application boundary to indicate an error occurred during processing, confirm that processing is complete or verify that processing should continue.

For example (EO/EQ), if a user tries to request a listing, but does not have access to the information, count one DET for the system response.

- Count one DET for the ability to specify an action to be taken even if there are multiple methods for invoking the same logical process.

For example (EO/EQ), if the user can initiate the generation of a report by clicking on the OK button or by pressing a PF key, count one DET for the ability to initiate the report.

- Do not count fields that are retrieved or derived by the system and stored on an ILF during the elementary process if the fields did not cross the application boundary.

  For example (EO), when a paycheck is printed, a status field on the employee ILF is updated to indicate that the check has been printed. Do not count the status field as a DET since it did not cross the boundary.

- Do not count literals as DETs.

For example (EO/EQ), literals include report titles, screen or panel identification, column headings, and field titles.

- Do not count paging variables or system-generated stamps.

  For example (EO/EQ), system-generated variables and stamps include
  - Page numbers
  - Positioning information such as "Rows 37 to 54 of 211"
  - Paging commands such as previous, next, and paging arrows on a GUI application
  - Date and time fields if they are displayed.

Hints to Help with Counting EIs, EOs and EQs

The following hints may help you apply the EI, EO and EQ counting rules.
Caution: The hints *are not* rules and should not be used as rules.

- Is data received from outside the application boundary?

  - Look at the work flow.
  - Identify where the user and other application interfaces occur in the process functional decomposition.
- Is the process the smallest unit of activity from the user perspective?
  - Look at the different paper or on-line forms used.
  - Review the ILFs to identify how the user groups the information.
  - Identify where the user and other application interfaces occur in the process functional decomposition.
  - Look at what happened in the manual system.
  - Note that one physical input or transaction file or screen can, when viewed logically, correspond to a number of EIs, EOs or EQs.
  - Note that two or more physical input or transaction files or screens can correspond to one EI, EO or EQ if the processing logic is identical.
- Is the process self-contained and does it leave the business in a consistent state?
  - Review other external inputs, external outputs and external inquiries to understand how the user works with the information.
  - Work through the process diagram to get hints.
  - Look at what happened in the manual system.
  - Check for consistency with other decisions.
- Is the processing logic unique from other EIs, EOs and EQs?
  - Identify batch inputs or outputs based on the processing logic required.
  - Remember that sorting or rearranging a set of data does not make processing logic unique.
- Are the data elements different from those for other EIs, EOs or EQs?
  - If the data elements appear to be a subset of the data elements of another EI, EO, or EQ, be sure two elementary processes are required by the user – one for the main data elements and one for the subsets.
- Identify the primary intent of the elementary process before classifying it as an EI, EO, or EQ.
- Identification of the elementary process(es) is based on a joint understanding or interpretation of the requirements between the user and the developers.
- Each element in a functional decomposition may not map to a unique elementary process.
- The identification of the elementary processes requires interpretation of the user requirements. Count only one FTR for each ILF/EIF referenced even if the ILF/EIF has multiple RETs.

Additional Hints to Help Counting EOs and EQs

- Is the process the smallest unit of activity from the user perspective?
  - An EO or EQ can be triggered by a process inside the application boundary. For example, the user requires that a report of all changed employee pay rates be sent to the budgeting area every 8 hours based on an internal clock. Situation A. The report contains employee name, SSN, and hourly pay rate which are all retrieved from the employee file. This is the smallest unit of activity from the user's perspective, contains no mathematical formulas or calculations, and no ILF is maintained in the process. This is one EQ. Situation B. The report contains employee name, SSN, and hourly pay rate which are all retrieved from the employee file. The report also includes the percentage pay change for the employee which is calculated from the data on the employee file. This is the smallest unit of activity from the user's

perspective, and no ILF is maintained in the process. However, since the process contains a mathematical formula, this is one EO.

- Derived data for an EO does not have to be displayed on the output. For example, each month, a report is generated listing all employees due for appraisal in the next 30 days. The records are selected by calculating next appraisal date based on the employee's last appraisal date, which is a field on the employee file, and the current date plus 30 days. This would be counted as one EO, and not as an EQ.

General System Characteristics

The Unadjusted Function Point count is multiplied by an adjustment factor called the Value Adjustment Factor (VAF). This factor considers the system's technical and operational characteristics and is calculated by answering 14 questions. The factors are:

1. DATA COMMUNICATIONS

The *data* and *control* information used in the application are sent or received over communication facilities. Terminals connected locally to the control unit are considered to use communication facilities. Protocol is a set of conventions which permit the transfer or exchange of information between two systems or devices. All data communication links require some type of protocol.

Score As:

0 Application is pure batch processing or a standalone PC.
1 Application is batch but has remote data entry *or* remote printing.
2 Application is batch but has remote data entry *and* remote printing.
3 Application includes online data collection or TP (teleprocessing) front end to a batch process or query system.
4 Application is more than a front-end, but supports only one type of TP communications protocol.
5 Application is more than a front-end, and supports more than one type of TP communications protocol.

2. DISTRIBUTED DATA PROCESSING

Distributed data or processing functions are a characteristic of the application within the application boundary.

Score As:

0 Application does not aid the transfer of data or processing function between components of the system.
1 Application prepares data for end user processing on another component of the system such as PC spreadsheets and PC DBMS.
2 Data is prepared for transfer, then is transferred and processed on another component of the system (not for end-user processing).
3 Distributed processing and data transfer are online and in one direction only.
4 Distributed processing and data transfer are online and in both directions.
5 Processing functions are dynamically performed on the most appropriate component of the system.

3. PERFORMANCE

Application performance objectives, stated or approved by the user, *in either* response or throughput, influence (or will influence) the design, development, installation, and support of the application.

Score As:

0 No special performance requirements were stated by the user.
1 Performance and design requirements were stated and reviewed but no special actions were required.
2 Response time or throughput is critical during peak hours. No special design for CPU utilization was required. Processing deadline is for the next business day.
3 Response time or throughput is critical during all business hours. No special design for CPU utilization was required. Processing deadline requirements with interfacing systems are constraining.
4 In addition, stated user performance requirements are stringent enough to require performance analysis tasks in the design phase.
5 In addition, performance analysis tools were used in the design, development, and/or implementation phases to meet the stated user performance requirements.

4. HEAVILY USED CONFIGURATION

A heavily used operational configuration, requiring special design considerations, is a characteristic of the application. For example, the user wants to run the application on existing or committed equipment that will be heavily used.

Score As:

0 No explicit or implicit operational restrictions are included.
1 Operational restrictions do exist, but are less restrictive than a typical application. No special effort is needed to meet the restrictions.
2 Some security or timing considerations are included.
3 Specific processor requirements for a specific piece of the application are included.
4 Stated operation restrictions require special constraints on the application in the central processor or a dedicated processor.
5 In addition, there are special constraints on the application in the distributed components of the system.

5. TRANSACTION RATE

The transaction rate is high and it influenced the design, development, installation, and support of the application.

Score As:

1 Peak transaction period (e.g., monthly, quarterly, seasonally, annually) is anticipated.
2 Weekly peak transaction period is anticipated.
3 Daily peak transaction period is anticipated.
4 High transaction rate(s) stated by the user in the application requirements or service level agreements are high enough to require performance analysis tasks in the design phase.

5 High transaction rate(s) stated by the user in the application requirements or service level agreements are high enough to require performance analysis tasks and, in addition, require the use of performance analysis tools in the design, development, and/or installation phases. Online data entry and control functions are provided in the application.

6. ONLINE DATA ENTRY

On-line data entry and control information functions are provided in the application.

Score As:

0 All transactions are processed in batch mode.
1 1% to 7% of transactions are interactive data entry.
2 8% to 15% of transactions are interactive data entry.
3 16% to 23% of transactions are interactive data entry.
4 24% to 30% of transactions are interactive data entry.
5 More than 30% of transactions are interactive data entry.

7. END-USER EFFICIENCY

The online functions provided emphasize a design for end-user efficiency. The design includes:

- Navigational aids (for example, function keys, jumps, dynamically generated menus)
- Menus
- Online help and documents
- Automated cursor movement
- Scrolling
- Remote printing (via online transactions)
- Pre-assigned function keys
- Batch jobs submitted from online transactions
- Cursor selection of screen data
- Heavy use of reverse video, highlighting, colors underlining, and other indicators
- Hard copy user documentation of online transactions
- Mouse interface
- Pop-up windows.
- As few screens as possible to accomplish a business function
- Bilingual support (supports two languages; count as four items)
- Multilingual support (supports more than two languages; count as six items)

Score As:

0 None of the above.
1 One to three of the above.
2 Four to five of the above.
3 Six or more of the above, but there are no specific user requirements related to efficiency.

4 Six or more of the above, and stated requirements for end user efficiency are strong enough to require design tasks for human factors to be included (for example, minimize key strokes, maximize defaults, use of templates).
5 Six or more of the above, and stated requirements for end user efficiency are strong enough to require use of special tools and processes to demonstrate that the objectives have been achieved.

8. ONLINE UPDATE

The application provides online update for the internal logical files.

Score As:

0 None.
1 Online update of one to three control files is included. Volume of updating is low and recovery is easy.
2 Online update of four or more control files is included. Volume of updating is low and recovery easy.
3 Online update of major internal logical files is included.
4 In addition, protection against data lost is essential and has been specially designed and programmed in the system.
5 In addition, high volumes bring cost considerations into the recovery process. Highly automated recovery procedures with minimum operator intervention are included.

9. COMPLEX PROCESSING

Complex processing is a characteristic of the application. The following components are present:

- Sensitive control (for example, special audit processing) and/or application specific security processing
- Extensive logical processing
- Extensive mathematical processing
- Much exception processing resulting in incomplete transactions that must be processed again, for example, incomplete ATM transactions caused by TP interruption, missing data values, or failed validations
- Complex processing to handle multiple input/output possibilities, for example, multimedia, or device independence

Score As:

0 None of the above.
1 Any one of the above.
2 Any two of the above.
3 Any three of the above.
4 Any four of the above.
5 All five of the above.

10. REUSABILITY

The application and the code in the application have been specifically designed, developed, and supported to be usable in *other* applications.

Score As:

0 No reusable code.
1 Reusable code is used within the application.
2 Less than 10% of the application considered more than one user's needs.
3 Ten percent (10%) or more of the application considered more than one user's needs.
4 The application was specifically packaged and/or documented to ease re-use, and the application is customized by the user at source code level.
5 The application was specifically packaged and/or documented to ease re-use, and the application is customized for use by means of user parameter maintenance.

11. INSTALLATION EASE

Conversion and installation ease are characteristics of the application. A conversion and installation plan and/or conversion tools were provided and tested during the system test phase.

Score As:

0 No special considerations were stated by the user, and no special setup is required for installation.
1 No special considerations were stated by the user *but* special setup is required for installation.
2 Conversion and installation requirements were stated by the user, and conversion and installation guides were provided and tested. The impact of conversion on the project is not considered to be important.
3 Conversion and installation requirements were stated by the user, and conversion and installation guides were provided and tested. The impact of conversion on the project is considered to be important.
4 In addition to 2 above, automated conversion and installation tools were provided and tested.
5 In addition to 3 above, automated conversion and installation tools were provided and tested.

12. OPERATIONAL EASE

Operational ease is characteristic of the application. Effective start-up, back-up, and recovery procedures were provided and tested during the system test phase. The application minimizes the need for manual activities, such as tape mounts, paper handling, and direct on-location manual intervention.

Score As:

0 No special operational considerations other than the normal back-up procedures were stated by the user.

1 - 4 One, some, or all of the following items apply to the application. Select all that apply. Each item has a point value of one, except as noted otherwise.

- Effective start-up, back-up, and recovery processes were provided, but operator intervention is required.
- Effective start-up, back-up, and recovery processes were provided, but no operator intervention is required (count as two items).
- The application minimizes the need for tape mounts.
- The application minimizes the need for paper handling.

5 The application is designed for unattended operation. Unattended operation means *no operator intervention* is required to operate the system other than to start up or shut down the application. Automatic error recovery is a feature of the application.

13. MULTIPLE SITES

The application has been specifically designed, developed, and supported to be installed at multiple sites for multiple organizations.

Score As:

0 User requirements do not require considering the needs of more than one user/installation site.
1 Needs of multiple sites were considered in the design, and the application is designed to operate only under identical hardware and software environments.
2 Needs of multiple sites were considered in the design, and the application is designed to operate only *under similar* hardware and/or software environments.
3 Needs of multiple sites were considered in the design, and the application is designed to operate *under different* hardware and/or software environments.
4 Documentation and support plan are provided and tested to support the application at multiple sites and the application is as described by 1 or 2.
5 Documentation and support plan are provided and tested to support the application at multiple sites and the application is as described by 3.

14. FACILITATE CHANGE

The application has been specifically designed, developed, and supported to facilitate change.
The following characteristics can apply for the application:

- Flexible query and report facility is provided that can handle simple requests; for example, *and/or* logic applied to only one internal logical file (count as one item).
- Flexible query and report facility is provided that can handle requests of average complexity, for example, *and/or* logic applied to more than one internal logical file (count as two items).
- Flexible query and report facility is provided that can handle complex requests, for example, *and/or* logic combinations on one or more internal logical files (count as three items).
- Business control data is kept in tables that are maintained by the user with online interactive processes, but changes take effect only on the next business day.
- Business control data is kept in tables that are maintained by the user with online interactive processes, and the changes take effect immediately (count as two items).

Score As:

0 None of the above.
1 Any one of the above.
2 Any two of the above.
3 Any three of the above.
4 Any four of the above.
5 All five of the above.

Adjusted FP Count

Each of these factors is scored based between 0-5 on their influence on the system being counted. The resulting score will increase or decrease the Unadjusted Function Point count by 35%. This calculation provides us with the Adjusted Function Point count.

Degree of Influence (DI) = sum of scores of 14 general system characteristics

Value Adjustment Factor (VAF) = 0.65 + DI / 100
The final Function Point Count is obtained by multiplying the VAF times the Unadjusted Function Point (UFP).

FP = UFP * VAF

Lecture No. 12

Software Process and Project Metrics

Everyone asks this question: how do I identify the problem? The answer is measure your process. Measurement helps in identification of the problem as well as in determining the effectiveness of the remedy.

Measurement is fundamental for providing mechanisms for objective evaluation of any process or activity. According to Lord Kelvin:

> *When you can measure what you are speaking about and express it in numbers, you know something about it; but when you cannot measure, when you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind: it may be the beginning of knowledge, but you have scarcely, in your thoughts, advanced to the stage of a science.*

In order to understand what your problems are, you need to measure. Only then a remedy can be applied. Take the example of a doctor. He measures and monitors different types of readings from a patient (temperature, heart beat, blood pressure, blood chemistry, etc) before proposing a medicine. After giving the medicine, the doctor monitors the effect of the medicine through the follow-up visits and makes the necessary adjustments. This process of measurement, correction and feedback is inherent in all kinds of systems. Software is no exception!

The idea is to measure your product and process to improve it continuously. Now the question is: how can we measure the quality of a software process and system?

Software project management primarily deals with metrics related to productivity and quality. For planning and estimation purposes, we look at the historic data – productivity of our team and the quality of their deliverables in the past projects. This data from the previous efforts is used to determine and estimate the effort required in our current project to deliver a product with a predictable quality. This data is also used to analyze the system bottlenecks and helps us in improving the productivity of our team and the quality of our product.

Measures, Metrics and Indicators

Before we can talk about the measurement process, we first need to understand the terms *measure*, *metrics*, and *indicators*. The terms *measure*, *measurement*, and *metrics* are often used interchangeable but there are significant differences among them. Within the software engineering domain, a measure provides a quantitative value of some attribute of a process or a product. For example, size is one measure of a software product. Measurement is the process or mechanism through which the measure is taken. For example, FP analysis is a mechanism to measure the size of software. Measurement involves taking one or more data points related to some aspect of the product or process. Software metric relates individual software measures to provide a normalized view. For example, defects per function point are one metric which relates two individual measures, that is, defects and size, into one metric.

Metrics give you a better insight into the state of the process or product. These insights are not the problems but just the *indicators* of problems. A software engineers collects measures and develops metrics and indicators.

Tom Gilb says, "Anything that you need to quantify can be measured in some way that is superior to not measuring at all!" This quote has two messages:

1. Anything can be measured
2. It is always better to measure than not doing it even if you do not have a good measuring device; it always gives you some information that you can use.

Metrics for software quality

The most important objective of any engineering activity is to produce high quality product with limited resources and time. The quality of the product cannot be determined if it is be measured.

The quality of the end result depends upon the quality of the intermediate work products. If the requirements, design, code, and testing functions are of high quality, then the chances are that the end product will also be of good quality. So, a good software engineer would adopt mechanisms to measure the quality of the analysis and design models, the source code, and the test cases.

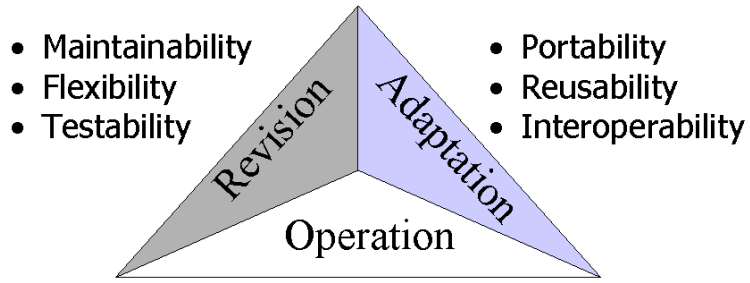
At the project level, the primary focus is to measure errors and defects and derive relevant metrics such as requirement or design errors per function point, errors uncovered per review hour, errors per thousand lines of code. These metrics provide an insight into the effectiveness of the quality assurance activities at the team as well as individual level.

Lecture No. 13

Software Quality Factors

In 1978, McCall identified factors that could be used to develop metrics for the software quality. These factors try to assess the inner quality of software from factors that can be observed from outside. The basic idea is that the quality of the software can be inferred if we measure certain attributes once the product is put to actual use. Once completed and implemented, it goes through three phases: operation (when it is used), during revisions (when it goes through changes), and during transitions (when it is ported to different environments and platforms).

During each one of these phases, different types of data can be collected to measure the quality of the product. McCall's model is depicted and explained as follows.



- Correctness
- Reliability
- Usability
- Integrity
- Efficiency

Factors related with operation

- Correctness
  - The extent to which a program satisfies its specifications and fulfills the customer's mission objectives

- Reliability
  - The extent to which a program can be expected to perform its intended function with required precision.

- Efficiency
  - The amount of computing resources required by a program to perform its function

- Integrity
  - Extent to which access to software or data by unauthorized persons can be controlled.

- Usability
  - Effort required to learn, operate, prepare input, and interpret output of a program

Factors related with revision

- Maintainability
  - Effort required to locate and fix an error in a program
- Flexibility
  - Effort required to modify an operational program
- Testability
  - Effort required to test a program to ensure that it performs its intended function

Factors related with adaptation

- Portability
  - Effort required transferring the program from one hardware and/or software system environment to another.
- Reusability
  - Extent to which a program can be reused in other applications
- Interoperability
  - Effort required to couple one system to another.

It is interesting to note that the field of computing and its theoretical have gone through phenomenal changes but McCall's quality factors are still as relevant as they were almost 25 years ago.

Measuring Quality

Gilb extends McCall's idea and proposes that the quality can be measured if we measure the correctness, maintainability, integrity, and usability of the product.

Correctness is defined as the degree to which software performs its function. It can be measured in defects/KLOC or defects/FP where defects are defined as verified lack of conformance to requirements. These are the problems reported by the user after release. These are counted over a standard period of time which is typically during the first year of operation.

Maintainability is defined as the ease with which a program can be corrected if an error is encountered, adapted if environment changes, enhanced if the customer requires an enhancement in functionality. It is an indirect measure of the quality.
A simple time oriented metric to gauge the maintainability is known as MMTC – mean time to change. It is defined as the time it takes to analyze the change request, design an appropriate modification, implement the change, test it, and implement it.

A cost oriented metric used to assess maintainability is called Spoilage. It is defined as the cost to correct defects encountered after the software has been released to the users. Spoilage cost is plotted against the overall project cost as a function of time to determine

whether the overall maintainability of software produced by the organization is improving.
Integrity is an extremely important measure especially in today's context when the system is exposed to all sorts to attacks because of the internet phenomenon. It is defined as software's ability to withstand attack (both accidental and intentional) to its security. It includes integrity of program, data, and documents. To measure integrity, two additional attributes are needed. These are: threat and security.

Threat is the probability (derived or measured from empirical evidence) that an attack of a specific type will occur within a given time and security is the probability that an attack of a specific type will be repelled. So the integrity of a system is defined as the sum of all the probability that the threat of a specific type will not take place and the probability that if that threat does take place, it will not be repelled.

$$\text{Integrity} = \sum [(1\text{-threat}) \times (1\text{-security})]$$

Finally, usability is a measure of user friendliness – the ease with which a system can be used. It can be measured in terms of 4 characteristics:

- Physical or intellectual skill required learn the system
- The time required to become moderately efficient in the use of system
- The net increase in productivity
- A subjective assessment

It is important to note that except for the usability, the other three factors are essentially the same as proposed by McCall.

Defect Removal Efficiency

Defect removal efficiency is the measure of how many defects are removed by the quality assurance processes before the product is shipped for operation. It is therefore a measure that determines the effectiveness of the QA processes during development. It is useful at both the project and process level.
Defect removal efficiency is calculated as the number of defect removed before shipment as a percentage of total defects

$$DRE = E/(E+D)$$

Where

- E – errors found before delivery
- D – errors found after delivery (typically within the first year of operation)

Regarding the effectiveness of various QA activities, Capers Jones published some data in 1997 which is summarized in the following table.

Design Inspection					●		●	●	●
Code Inspection				●		●		●	●
Quality Assurance			●			●	●		●
Testing		●	●	●	●	●	●	●	●
Worst	**30%**	**37%**	**50%**	**55%**	**65%**	**75%**	**77%**	**85%**	**95%**
Median	**40%**	**53%**	**65%**	**70%**	**80%**	**87%**	**90%**	**97%**	**99%**
Best	**50%**	**60%**	**75%**	**80%**	**87%**	**93%**	**95%**	**99%**	**99.9%**

In this research, they tried to measure the effectiveness of 4 different activities namely design inspection, code inspection, quality assurance function, and testing. It is important to note that testing alone only yields a DRE of 40% on the average. However, when it is combined with design and code inspection, the DRE reaches 97%. That means, code and design inspection are extremely important activates that are unfortunately not given their due importance.

Lecture No. 14

Metrics for specification quality

As mentioned earlier, the quality of the software specification is of extreme importance as the entire building of software is built on this foundation. A requirement specification document is measured in terms of lack of ambiguity, completeness, consistency, correctness; understand ability, verifiability, achievability, concision, traceability, modifiability, precision, and reusability.

Metrics to assess the quality of the analysis models and the corresponding software specification were proposed by Davis in 1993 for these seemingly qualitative characteristics.

For example, the numbers of requirements are calculated as:

$$n_r = n_f + n_{nf}$$

where
n_r – total number of requirements
n_f – functional requirements
n_{nf} – non-functional requirements

Now lack of ambiguity in the requirements is calculated as:

$$Q_1 = n_{ui}/n_r$$

Where
n_{ui} – number of requirements for which all reviewers had identical interpretation (i.e. unambiguous requirements)

Similarly, completeness is measures as follows:

$$Q_2 = n_u/ [n_i \times n_s]$$

n_u – unique functional requirements
n_i – number of inputs (stimuli)
n_s – number of states specified

On the similar lines, the quality of design is also measured quantitatively.
The quality of the architectural design can be measured by measuring its complexity as shown below:

- Structural complexity $S = (f_{out})^2$
- Data complexity $D = v/(f_{out} + 1)$
  - 'v' is the number of input and output variables
- System complexity $C = \sum (S_i + D_i)$

Baseline

In order to use the data for estimation and drawing conclusions, it must be base-lined. In the baseline, data from past projects is collected, cleaned, and put in a database. Such metrics baseline is used to reap benefits at the process, project, and product level.

As mentioned above, the metrics baseline consists of data collected from past project over a period of time. To be effective, the data must be reasonably accurate, it should be collected over many projects, measures must be consistent – same technique or yardstick for data collection should have been used, applications should be similar to work that is to be estimated, and feedback to improve baseline's quality.

Metrics for small organizations

The metric program can be quite complex and extensive. Small organization would find it difficult to implement a full-fledged metrics program as it may require quite a number of resources. However, it is important to appreciate that a metrics program of even a smaller scale would also be of immense value and therefore all organizations of all sizes should have should have one. It can be a very simple and can be implemented with the help of simple and inexpensive tools and methods.

Small organizations – around 20 or so people – must measure in a cost effective manner. In order for it to be effective, it should be simple and value-oriented and should focus on result rather than measurement. It is important to establish the objectives of measurement.

This is illustrated by the following example.

Let us assume that we wanted to reduce the time to evaluate and implement change requests in our organization. In order to achieve this objective, we needed to measure the following:

- Time (in hours or days) elapsed from the time a request is made until evaluation is complete - t_{queue}
- Size (fp) of the change request
- Effort (in person months) to perform the evaluation- W_{eval}
- Time elapsed from completion of evaluation to assignment of change order – t_{eval}
- Effort required to make the change – W_{change}
- Time required to make the change – t_{change}
- Errors uncovered during work to make change – E_{change}
- Defects uncovered after change is released – D_{change}

This data was then collected and stored in a simple database as shown below.

Project	Size (FP)	Effort (Pm)	Cost Rs. (000)	Pages of documentation	Pre-shipment errors	Post-shipment defects	People
abc	120	24	168000	365	134	29	3
def	270	62	440000	1224	321	86	5
ghi	200	43	314000	1050	256	64	6

This data is then normalized on per function point basis as follows:

Project	Size (FP)	Effort (Pm)	Cost Rs. (000)	Pages of documentation	Pre-shipment errors	Post-shipment defects	People
abc	120	0.2	1400	3.04	1.12	0.24	3
def	270	0.23	1629	4.53	1.19	0.32	5
ghi	200	0.22	1570	5.25	1.28	0.32	6

We are now ready to use this data to analyze the results of process changes and their impact on the time to implement change requests.

In order to do that, we need to employ some statistical techniques and plot the result graphically. This is known as statistical control techniques.

Lecture No. 15

Statistical Control Techniques – control charts

Same process metrics vary from project to project. We have to determine whether the trend is statistically valid or not. We also need to determine what changes are meaningful. A graphical technique known as control charts is used to determine this.

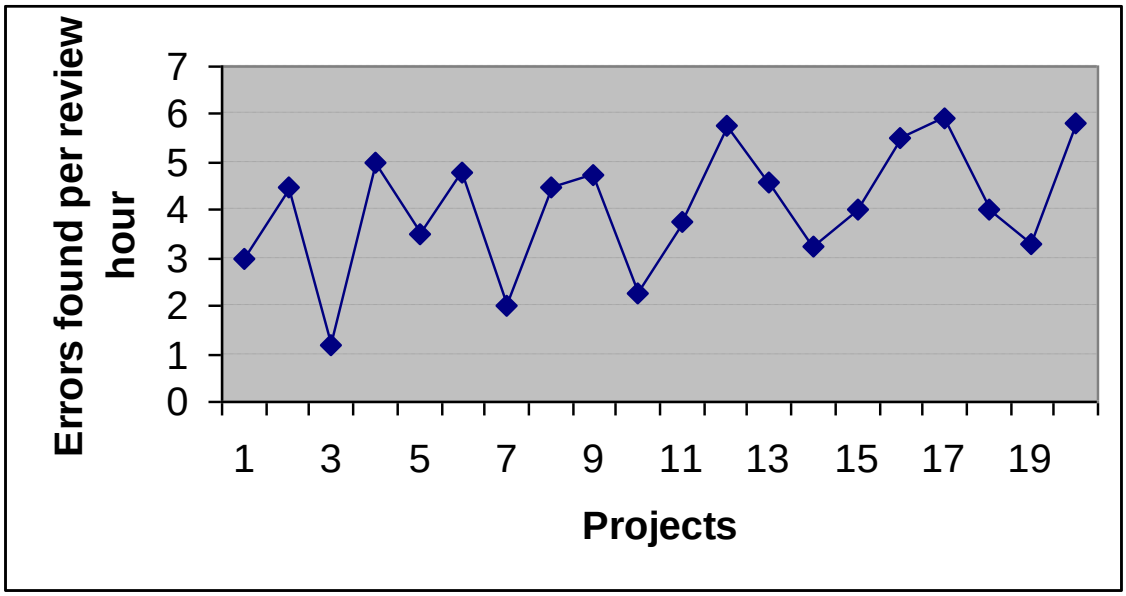
This technique was initially developed for manufacturing processes in the 1920's by Walter Shewart and is still very applicable even in disciples like software engineering. Control charts are of two types: moving range control chart and individual control chart. This technique enables individuals interested in software process improvement to determine whether the dispersion (variability) and "location" (moving average) of process metrics are stable (i.e. the process exhibits only natural or controlled changes) or unstable (i.e. the process exhibits out-of-control changes and metrics cannot be used to predict performance).

Let us now demonstrate the use of these control charts with the help of an example.
Let us assume that the data shown in the following table regarding the average change implementation time was collected over the last 15 months for 20 small projects in the same general software domain 20 projects. To improve the effectiveness of reviews, the software organization provided training and mentoring to all project team members beginning with project 11.

Project	Time /change implementation
1	3
2	4.5
3	1.2
4	5
5	3.5
6	4.8
7	2
8	4.5
9	4.75
10	2.25
11	3.75
12	5.75
13	4.6
14	3.25
15	4
16	5.5
17	5.9
18	4
19	3.3
20	5.8

In order to determine whether our change in process had any impact, we use control charts.

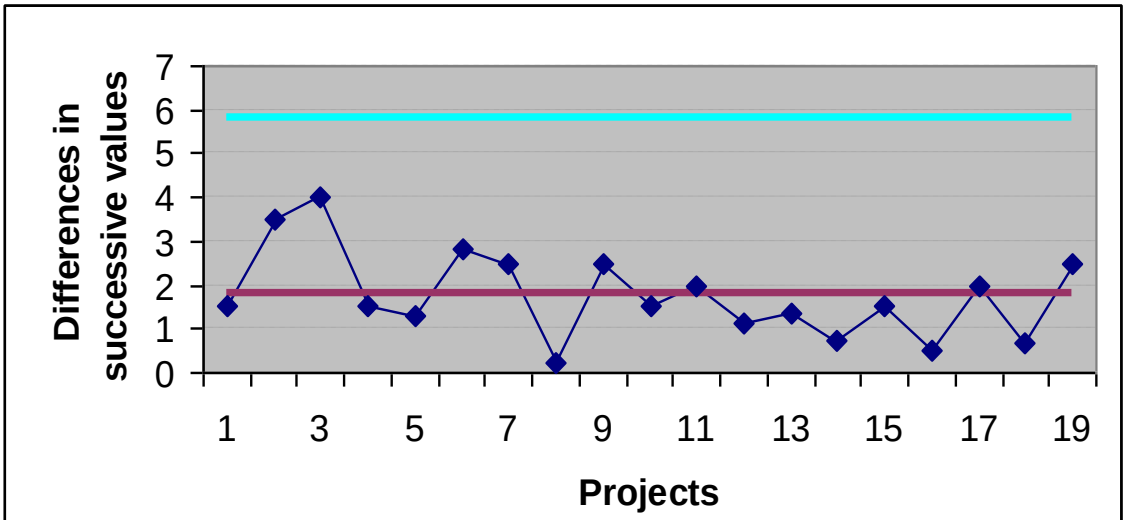
This data is first presented in the graph form as follows:



We now develop the Moving Range Control Chart as follows:

1. Calculate the moving ranges: the absolute values of the successive differences between each pair of data point. Plot these moving ranges on your chart. (the dark blue line)
2. Calculate the mean of the moving ranges. Plot this on the chart. (the red line)
3. Multiply the mean with 3.268. Plot this as the Upper Control Line (UCL). This line is 3 standard deviations above the line. (the light blue line)
4. To determine whether the process metrics description is stable, a simple question is asked: are the moving ranges values inside the UCL? If the answer is yes then the process is stable otherwise it is unstable.

This chart is shown in the following diagram:



This chart is then used to develop the individual control chart as follows:

1. Plot individual metric values
2. Compute the average value for the metrics values - A_m
3. Multiply the mean of moving average by 2.66 and add average computed in step 2 above. The result is Upper Natural Process Limit (UNPL)
4. Multiply the mean of moving average by 2.66 and subtract average computed in step 2 above. The result is Lower Natural Process Limit (LNPL)
5. Plot UNPL and LNPL. If LNPL is less than zero than it need not be plotted unless the metric being evaluated takes on values that are less than 0.
6. Compute a standard deviation as (UNPL – A_m)/3.
7. Plot lines 1 and 2 standard deviations above and below A_m.
8. Applying these steps we derive an individual control chart.

This chart may be used to evaluate whether the changes represented by metrics indicate a process that is in control or out of control. For this, the following 4 criteria zone rules are used.

1. A single metrics value lies outside UNPL
2. Two out of three successive values lay more than 2 standards deviations away from A_m.
3. Four out of five successive values lay more than one standard deviation away.
4. Eight consecutive values lie on one side of A_m.

If any of these tests passes, the process is out of control otherwise the process is within control.

Since none of the test passes for the data in our example, our process is in control and this data can be used for inference.

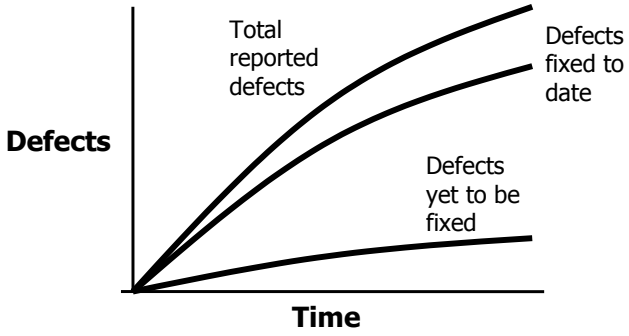
We now analyze our results. It can be seen that the variability decreased after project 10. By computing the mean value of the first 10 and last 10 projects, it can be shown that the remedial measure taken was successful and resulted in 29% improvement in efficiency of the process. Hence the process changes incorporated were useful and bore fruit.

Lecture No. 16

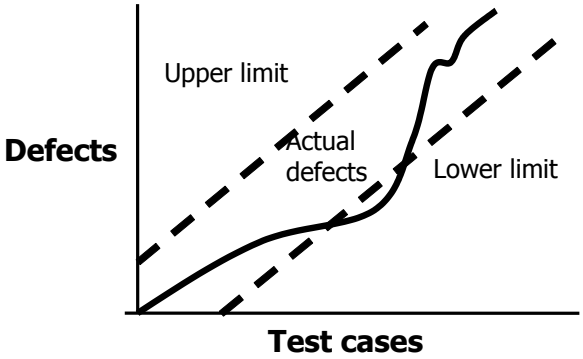
Interpreting Measurements

A good metric system is the one which is simple and cheap and at the same time adds a lot of value for the management. Following are some of the examples that can be used for effective project control and management.

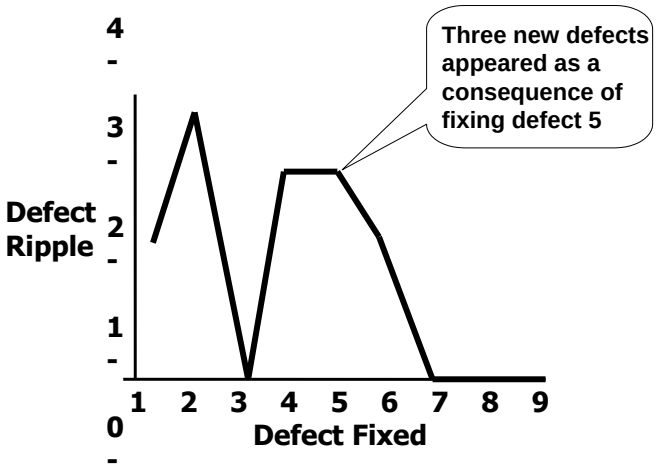
We can collect data about the defects reported, and defects fixed and plot them in the following manner, with their difference showing the defects yet to be fixed. This can give us useful information about the state of the product. If the gap between the defects reported and defects fixed is increasing, then it means that the product is in unstable condition. On the other hand if this gap is decreasing then we can say that the product is in a stable condition and we can plan for shipment.



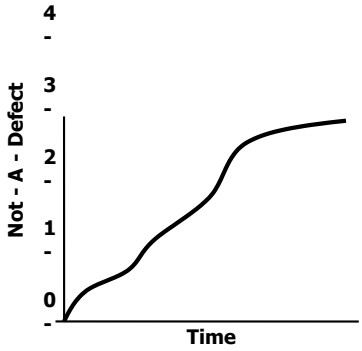
Similarly, we can gain useful information by plotting the defects reported against the number of use cases run. We can use control lines from our previous data and see if the actual defects are within those control limits. If the defects at any given point in time are less than the lower limit then it may mean that out testing team is not doing a good job and coverage is not adequate. On the other hand, if it crosses the upper line then it indicates that the design and coding is not up to mark and we perhaps need to check it.



Another very simple graph as shown below can give a lot of insight into the design quality. In this case, if the frequency of ripple defects is too large, then it means that then there is tight coupling and hence the design is not maintainable.



The following is yet another very simple and effective way of getting insight into the



quality of the requirements. If a number of defects that are reported by the testing team are ultimately resolved as not-a-defect then there may be a sever problem with the requirements document as two teams (development and testing) are interpreting it differently and hence coming to different conclusions.

Lecture No. 17

Software Project Planning

Software project planning is an activity carried out by the project manager to estimate and address the following points:

1. Software scope estimation
2. Resources requirements
3. Time requirements
4. Structural decomposition
5. Risk analysis and planning

Software scope estimation
Software scope describes the data and control to be processed, function, performance, constraints, interfaces, and reliability. Determination of the software scope is a pre-requisite of all sorts of estimates, including, resources, time, and budget.

In order to understand the scope, a meeting with the client should be arranged. The analyst should start with developing a relationship with the client representative and start with context-free questions. An understanding of the project background should also be developed. This includes understanding:

- Who is behind the request (sponsor)?
- Who will use the solution (users)?
- What are the economic benefits (why)?

Now is the time to address the find out the more about the product. In this context, the following questions may be asked:

- How would you characterize good output?
- What problems will the solution address?
- Can you show me the environment in which the solution will be used?
- Will any special performance issues or constraints affect the way the solution is approached?
- Are you the right person to answer these questions? Are answers “official”?
- Are my questions relevant to the problem that you have?
- Am I asking too many questions?
- Can anyone else provide additional information?
- Should I be asking you anything else?

In this regards, a technique known as **Facilitated Application Specification Techniques** or simply **FAST** can be used. This is a team-oriented approach to requirement gathering that is used during early stages of analysis and specification. In this case joint team of customers and developers work together to identify the problem, propose elements of the solution, negotiate different approaches, and specify a preliminary set of requirements.

Feasibility

The purpose of the feasibility analysis is to determine can we build software to meet the scope? For this purpose, the project is analyzed on the following 4 dimensions:

Technology

- Is the project technically feasible?
- Is it within the state of the art?
- Can defects be reduced to a level matching the application needs?

Finance

- Is it financially feasible?
- Can development be completed at a cost that software organization, its client, or the market can afford?

Time

- Will the project time to market beat the competition?
- Can we complete the project in the given amount of time?

Resources

- Does the organization have resources needed to succeed? The resources include:
  - HW/SW tools
  - Reusable software components
  - People

Software Project Estimation

Once the project feasibility has been determined, the project manager starts the estimation activity. It is a relatively difficult exercise and has not become an exact science. It is influenced by human, technical, environmental, political factors.

For software project estimation, a project manager can use historic data about its organizations previous projects, decomposition techniques, and/or empirical models developed by different researchers.

Empirical Models

Empirical models are statistical models and are based upon historic data. Although there are many different models developed by different researchers, all of them share the following basic structure:

$E = A + B * (ev)^C$

Where

A, B, c are empirical constants,
'ev' is the effort in terms of lines of code or FP, and
'E' is the effort in terms of person months.

The most famous of these models is the COCOMO - COnstructive COst MOdel – model. It also has many different versions. The simplest of these versions is given below:

$$E = 3.2 (KLOC)^{1.05}$$

Some of these models take into account the project adjustment components including problem complexity, staff experience, and development environment.

It is important to note that there are a number of models with each yielding a different result. This means that any model must be calibrated for local needs before it can be effectively used.

The Software Equation

The software equation shown below is dynamic multivariable estimation model. It assumes a specific distribution of effort over the life of the software development project and is derived from productivity data collected for over 4000 projects.

$$E = [LOC \times B^{0.333}/P]^3 \times (1/t^4)$$

Where:

- E – effort in person months or person years
- t – project duration in months or years
- B – special skill factor
  - Increases slowly as the need for integration, testing, QA, documentation, and management skills grow
- P – productivity parameter
  - Overall process maturity and management practices
  - The extent to which good SE practices are used
  - The level of programming language used
  - The state of the software environment
  - The skills and experience of the software team
  - The complexity of the application

Buy versus build

It is often more cost-effective to acquire than to build. There may be several different options available. These include:

- Off-the-shelf licensed software
- Software components to be modified and integrated into the application
- Sub-contract

The final decision depends upon the criticality of the software to be purchased and the end cost. The buy versus build decision process involves the following steps:

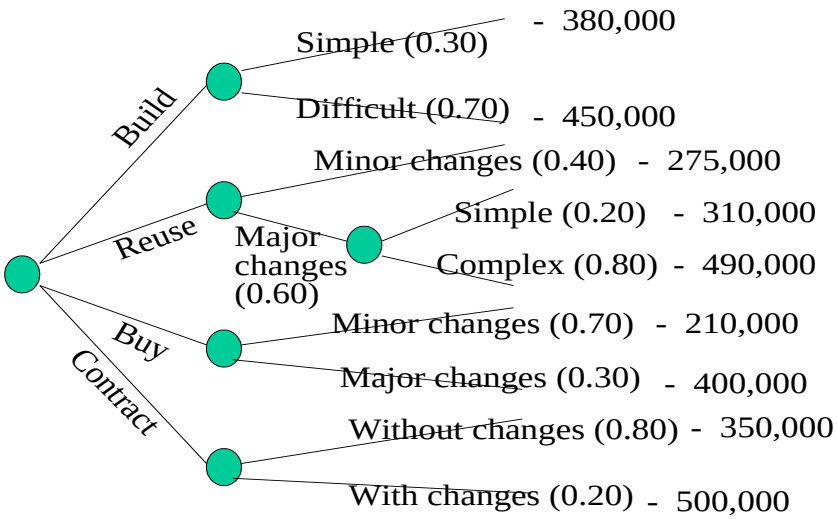
- Develop specification for function and performance of the desired software. Define measurable characteristics whenever possible.
- Estimate internal cost and time to develop
- Select 3-4 candidate applications that best meet your specifications
- Select reusable software components that will assist in constructing the required application
- Develop comparison matrix that presents a head-to-head comparison of key function. Alternatively, conduct benchmark tests to compare candidate software.

- Evaluate each software package or component based on past product quality, vendor support, product direction, reputation, etc.
- Contact other users of the software and ask for opinion.

The following key considerations must always be kept in the perspective:

- Delivery date
- Development Cost
  - Acquisition + customization
- Maintenance Cost

A decision tree can be built to analyze different options in this regards. As an example of this, let us consider the following. In this case, we have four different options, namely, build, reuse, buy, and contract. We analyze each one of these with the help of a decision tree. Each node of the tree is further partitioned a probability is assigned to each branch. At the end, cost for each path in the tree, from root to a leaf, is estimated and associated with that path. This process is shown in the following diagram.



Once the information in the tree is complete, it is used to determine the most viable option. For each option the expected cost is determined as follows:

Expected cost = $\sum$ (path probability)$_i$ x (estimated path cost)

Using this formula, we calculate the expected cost of each option as follows:

Build = 0.30*380 + 0.70*450000 = 429000
Reuse = 0.4*275000 + 0.6*0.2*310000 + 0.6*0.8*490000 = 382000

The expected cost of Buy and Contract can also be calculated in a similar fashion and comes out to be:

Buy = 267000
Contract = 410000

Therefore, according to this analysis, it is most viable to buy than any other option.

Lecture No. 18

Risk analysis and management

Analysis and management of project risks is also a very important activity that a project manager must perform in order to improve the chances for the project. Robert Charette defines risk as follows:

> *Risk concerns future happenings. Today and yesterday are beyond active concern. The question is, can we, therefore, by changing our action today create and opportunity for a different and hopefully better situation for ourselves tomorrow. This means, second, that risk involves change, such as changes in mind, opinion, action, or places. … [Third,] risks involve choice, and the uncertainty that choice itself entails.*

Risk analysis and management involves addressing the following concerns:

1. Future – what risks might cause the project to go awry
2. Change – what change might cause the risk to strike
    - How changes in requirements, technology, personnel and other entities connected to the project affect the project
3. Choice – what options do we have for each risk

In Peter Drucker words:

> *while it is futile to try to eliminate risk, and questionable to try to minimize it, it is essential that the risk taken be the right risk.*

There are two basic risk management philosophies, reactive and proactive.

- Reactive – Indiana Jones school of risk management
    - Never worrying about problems until they happened, and then reacting in some heroic way – Indiana Jones style.
- Proactive
    - Begins long before technical work starts
    - Risks are identified, their probability and impact are analyzed, and they are ranked by importance.
    - Risk management plan it prepared
        - Primary objective is to avoid risk
        - Since all risks cannot be avoided, a contingency plan is prepared that will enable it to respond in a controlled and effective manner

Unfortunately, Indiana Jones style is more suitable for fiction and has a rare chance of success in real life situations. It is therefore imperative that we manage risk proactively.

A risk has two characteristics:

- Uncertainty – the risk may or may not happen
- Loss – if the risk becomes a reality, unwanted consequences or losses will occur.

A risk analysis involves quantifying degree of uncertainty of the risk and loss associate with it. In this regards, the PM tries to find answers to the following questions:

- What can go wrong?
- What is the likelihood of it going wrong?
- What will the damage be?
- What can we do about it?

© Virtual University of Pakistan
thegradup.com

Lecture No. 19

Types of Risks

Each project is faced with many types of risks. These include:

- Project risks
  - Will impact schedule and cost
  - Includes budgetary, schedule, personnel, resource, customer, requirement problems
- Technical risks
  - Impact the quality, timelines, and cost
  - Implementation may become difficult or impossible
  - Includes design, implementation, interface, verification and maintenance problems
  - Leading edge technology
- Business risks
  - Marketability
  - Alignment with the overall business strategy
  - How to sell
  - Losing budget or personnel commitments

Furthermore, there are predictable and unpredictable risks. Predictable risks can be uncovered after careful evaluation whereas unpredictable risks cannot be identified.

Risk Identification

It is the responsibility of the project manager to identify known and predictable risks. These risks fall in the following categories of generic risks and product specific risks. *Generic risks* are threats to every project whereas *Product specific risks* are specific to a particular project. The question to be asked in this context is: what special characteristics of this project may threaten your project plan? A useful technique in this regards is the preparation of a risk item checklist. This list tries to ask and answer questions relevant to each of the following topics for each software project:

- Product size
- Business impact
- Customer characteristics
- Process definition
- Development environment
- Technology to be built
- Staff size and experience

Assessing Overall Project Risks

In order to assess the overall project risks, the following questions need to be addressed:

- Have top software and customer managers formally committed to support the project?
- Are end-users committed to the project and the system/product to be built?
- Are requirements fully understood?
- Have customers been involved fully in requirement definition?
- Do end-users have realistic expectations?
- Does the software team have right mix of skills?
- Are project requirements stable?
- Does the project team have experience with the technology to be implemented?
- Is the number of people on the project team adequate to do the job?

Risk components and drivers

Each risk has many components and forces behind them. From this perspective, risks can be categorized into the following categories:

- Performance risks
  - Degree of uncertainty that the product will meet its requirements and be fit for its intended use
- Cost risks
  - The degree of uncertainty that the project budget will be maintained
- Support risks
  - Resultant software will be easy to correct, enhance, and adapt
- Schedule risks
  - Product schedule will be maintained

Each risk has its own impact and can be characterized as negligible, marginal, critical, or catastrophic.

This is summarized in the following table:

Risk Impact		**Performance**	**Support**	**Cost**	**Schedule**
Catastrophic	Consequence of error	Failure to meet the requirements will result in mission failure		Results in increased cost and schedule delays. Expected value in excess of $500K	
	Consequence of failure to achieve desired result	Significant degradation	Non-responsive or unsupportable	Budget overrun likely	Unachievable
Critical	Consequence of error	Would degrade performance to a point where mission success is questionable		Results in operational delays and or increased cost with expected value of $100K-$500K	
	Consequence of failure to achieve desired result	Some reduction in technical performance	Minor delays	Possible overrun	Possible slippage

Marginal	Consequence of error	Result in degradation of secondary mission		Expected value <$100K	
	Consequence of failure to achieve desired result	Small reduction	Responsive	Sufficient financial resources	Realistic
Negligible	Consequence of error	Inconvenience		Minor	
	Consequence of failure to achieve desired result	No reduction	Supportable	Budget under run possible	achievable

Risk Projection

Risk projection is concerned with risk estimation. It attempts to rate risks in two ways: likelihood and consequences. There are four risk project activities. These are:

- Establish a scale that reflects the perceived likelihood of risk
- Delineate the consequences
- Estimate impact
- Note the overall accuracy of risk projection

This process is exemplified with the help of the following table:

Risk	Category	Probability	Impact	RMMM
Size estimate may be significantly low	PS	60%	2	
Larger number of users than planned	PS	30%	3	
Less reuse than planned	PS	70%	2	
End-users resist system	BU	40%	3	
Delivery deadline will be tightened	BU	50%	2	
Funding will be lost	CU	40%	1	
Customer will change requirements	PS	80%	2	
Technology will not meet expectations	TE	30%	1	
Lack of training on tools	DE	80%	3	
Staff inexperienced	ST	30%	2	
Staff turnover will be high	ST	60%	2	

Where impacts are codified as follows:
1: catastrophic 2: critical 3: marginal 4: negligible

and RMMM stands for risk mitigation, monitoring, and management plan.

Lecture No. 20

Assessing Risk Impact

Assessment of risk impact is a non-trivial process. Factors affecting the consequences are the nature, scope, and timing.

For each risk an exposure is calculated as follows:

RE = Probability of the risk x Cost

This risk exposure is then used to identify the top risks and mitigation strategies.

As an example, let us consider the following case:

- Risk:
  - Only 70% of the software components scheduled for reuse will, in fact, be integrated into the application. The remaining functionality will have to be custom developed.
- Risk Probability – 80% likely (i.e. 0.8)
- Risk impact
  - 60 reusable software components were planned. If only 70% can be used, 18 components would have to be developed from scratch. Since the average component will cost 100,000, the total cost will be 1,800,000.
- Therefore, RE = 0.8 * 1,800,000 = 1,440,000

A high level risk may be refined into finer granularity to handle it efficiently. As an example, the above mentioned risk is refined as follows:

1. Certain reusable components were developed by 3rd party with no knowledge of internal design standard.
2. Design standard for component interfaces has not been solidified and may not conform to certain existing components.
3. Certain reusable components have been implemented in a language that is not supported on the target environment.

We can now take the following measures to mitigate and monitor the risk:

1. Contact 3rd party to determine conformance with design standards.
2. Press for interface standard completion; consider component structure when deciding on interface protocol.
3. Check to determine if language support can be acquired.

This leads us to the following Management/Contingency Plan:

1. RE computed to 1,440,000. Allocate this amount within project contingency cost.
2. Develop revised schedule assuming 18 additional components will have to be custom-built
3. Allocate staff accordingly

Risk Mitigation, Monitoring, and Management (RMMM)

The RMMM plan assists the project team in developing strategy for dealing with risk. In this context, an effective strategy must consider:

- Risk avoidance
- Risk monitoring
- Risk management and contingency plan

It must always be remembered that avoidance is always the best strategy.

As an example, let consider the following scenario. In this case high turn-over has been identified as a risk with the following characteristics:

- Risk r_j - High turnover
- Likelihood $l_j = 0.7$
- Impact x_j - projected at level 2 (critical)

Let us now devise a mitigation strategy for reducing turnover. In order to do so, the following steps may be taken:

- Meet with current staff to determine causes for turnover (e.g. poor working conditions, low pay, competitive job market)
- Mitigate those causes that are under our control before the project starts
- Once the project commences, assume turnover will occur and develop techniques to ensure continuity when people leave
- Organize project teams so that information about each development activity is widely dispersed
- Define documentation standards and establish mechanisms to be sure that documents are developed in a timely manner (to ensure continuity)
- Conduct peer reviews of all work (so that more than one person is up to speed)
- Assign a backup staff member for every critical technology

Once the strategy has been devised, the project must be monitored for this particular risk. That is, we must keep an eye on the various factors that can indicate that this particular risk is about to happen. In this case, the factors could be:

- General attitude of team members based on project pressures
- The degree to which the team is jelled
- Interpersonal relationships among team members
- Potential problems with compensation and benefits
- The availability of jobs within the company and outside it

Also, the effectiveness of the risk mitigation steps should be monitored. So, in this example, the PM should monitor documents carefully to ensure that each can stand on its own and that each imparts information that would be necessary if a newcomer were forced to join the software team somewhere in the middle of the project.

Risk Management and Contingency Plan

Risk management and contingency planning assumes that mitigation efforts have failed and that the risk has become a reality.

- Risk has become a reality – some people announce that they will be leaving
- If mitigation strategy has been followed, backup is available, information has been documented, and knowledge has been dispersed
- Temporarily refocus and readjust resources
- People who are leaving are asked to stop all work and ensure knowledge transfer

Risk mitigation and contingency is a costly business. It is therefore important to understand that for RMMM plan, a cost/benefit analysis of each risk must be carried out. The Pareto principle (80-20 rule) is applicable in this case as well – 20% of the identified risk account for 80% of the potential for project failure.

Lecture No. 21

Software Project Scheduling and Monitoring

Software project scheduling is the next task to be performed by the PM. It is important to note once again that in the reasons for project failure, unrealistic deadline and underestimate of effort involved in the project are two of the most important reasons for project failure. Therefore, a good schedule estimate would increase the chances of the success of the project.

In this context, a PM has to first come up with the schedule and then monitor the progress of the project to ensure that things are happening according to the schedule. It would not be out of place to quote Fred Brooks at this point. He says, *"Projects fall behind schedule one day at a time."* That means a delay of a week or a month or a year does not happen suddenly – it happens one day at a time. Therefore, a project manager has to be vigilant to ensure that the project does not fall behind schedule.

The reality of a technical project is that hundreds of small tasks must occur to accomplish a large goal. Therefore the Project manager's objectives include:

- Identification and definition all project tasks
- Building a network that depicts their interdependencies
- Identification of the tasks that are critical within the network
- Tracking their progress to ensure delay is recognized one day at a time

For this, the schedule must be fine grained.

Software Project Scheduling

Software project scheduling is an activity that distributes estimated effort across the planned project duration by allocating the effort to specific software engineering tasks.

It is important to note that the schedule evolves over time. During early stages of project planning, a macroscopic schedule is developed. This type of schedule identifies all major SE activities and the product functions to which they are applied. As the project gets underway these tasks are refined into a detailed schedule.

In order to come up with a realistic schedule, the following basic principles are used:

- Compartmentalization

The project must be compartmentalized into a number of manageable activities and tasks. To accomplish compartmentalization, both the product and process are decomposed.

- Interdependency

The interdependency of each compartmentalized activity or task must be determined. Some tasks must occur in sequence while others can occur in parallel. Some activities cannot commence until the work product produced by another is available.

- Time allocation

Each task to be scheduled must be allocated some number of work units (e.g. person-days of effort). In addition, each task must be assigned a start date and an end date which is a function of the interdependencies and number of resources.

- Effort validation

Every project has a defined number of staff members. As time allocation occurs, the project manager must ensure that no more than the allocated number of people has been scheduled at any given time.

- Defined responsibilities

Every task should be assigned to a specific team member.

- Defined outcomes

Every task should have a defined outcome, normally a work product.

- Defined milestones

Every task or group of tasks should be associated with a project milestone.

Software Project Scheduling and Monitoring

Software project scheduling is the next task to be performed by the PM. It is important to note once again that in the reasons for project failure, unrealistic deadline and underestimate of effort involved in the project are two of the most important reasons for project failure. Therefore, a good schedule estimate would increase the chances of the success of the project.

In this context, a PM has to first come up with the schedule and then monitor the progress of the project to ensure that things are happening according to the schedule. It would not be out of place to quote Fred Brooks at this point. He says, *"Projects fall behind schedule one day at a time."* That means a delay of a week or a month or a year does not happen suddenly – it happens one day at a time. Therefore, a project manager has to be vigilant to ensure that the project does not fall behind schedule.

The reality of a technical project is that hundreds of small tasks must occur to accomplish a large goal. Therefore the Project manager's objectives include:

- Identification and definition all project tasks
- Building a network that depicts their interdependencies
- Identification of the tasks that are critical within the network
- Tracking their progress to ensure delay is recognized one day at a time

For this, the schedule must be fine grained.

Software Project Scheduling

Software project scheduling is an activity that distributes estimated effort across the planned project duration by allocating the effort to specific software engineering tasks.
It is important to note that the schedule evolves over time. During early stages of project planning, a macroscopic schedule is developed. This type of schedule identifies all major

SE activities and the product functions to which they are applied. As the project gets underway these tasks are refined into a detailed schedule.

In order to come up with a realistic schedule, the following basic principles are used:

- Compartmentalization

The project must be compartmentalized into a number of manageable activities and tasks. To accomplish compartmentalization, both the product and process are decomposed.

- Interdependency

The interdependency of each compartmentalized activity or task must be determined. Some tasks must occur in sequence while others can occur in parallel. Some activities cannot commence until the work product produced by another is available.

- Time allocation

Each task to be scheduled must be allocated some number of work units (e.g. person-days of effort). In addition, each task must be assigned a start date and an end date which is a function of the interdependencies and number of resources.

- Effort validation

Every project has a defined number of staff members. As time allocation occurs, the project manager must ensure that no more than the allocated number of people has been scheduled at any given time.

- Defined responsibilities

Every task should be assigned to a specific team member.

- Defined outcomes

Every task should have a defined outcome, normally a work product.

- Defined milestones

Every task or group of tasks should be associated with a project milestone.

Lecture No. 22

Relationship between people and effort

The relationship between the number of people and time to develop an application is not linear. It is not as simple as a 120 man-day project can be developed by 1 person working for 120 days or 120 people working for 1 day. The communication and coordination overhead plays a very significant role.

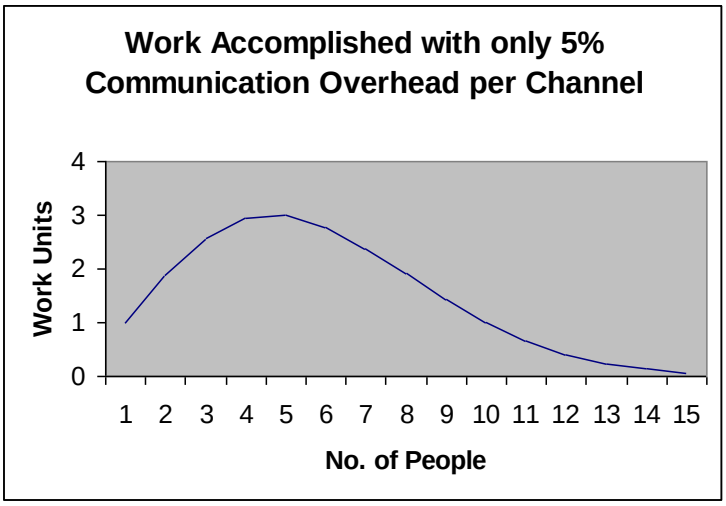
As can be recalled from our earlier discussions, total number of Channels of communication involving N people is given by the following formula.

$$C = N(N-1)/2$$

Now, if the communication overhead per channel is k, then work accomplished is given by:

$$W = (1-k)^C \times N$$

This phenomenon is depicted in the following diagram:



It may be noted here that with only a 5% communication overhead per channel, the total work accomplished by a team of 6 people would be less than the volume of work completed by a team of 4 people. It is also interesting to note that it approaches 0 as the team size approaches 20.

Task Set Definition

A process model defines a task set which comprises of SE work tasks, milestones, and deliverables. This enable a software team to define, develop, and support the software.

Therefore, each software process should define a collection of task sets, designed to meet the needs of different types of projects. To determine the set of tasks to be performed the type of the project and the degree of rigor required needs to be established. Different

types of projects and different degree of rigor. These projects could fall into the following categories:

- Concept development projects
- New application development
- Application enhancement
- Application maintenance
- Reengineering projects

The degree of rigor can also be categorized as Casual, Structured, Strict, or Quick Reaction. The following paragraphs elaborate each one of these.

- Casual
  All process framework activities are applied, but only a minimum task set is required. It requires reduced umbrella tasks and reduced documentation. Basic principles of SE are however still followed.
- Structured
  In this case a complete process framework is applied. Appropriate framework activities, related tasks, and umbrella activities (to ensure high quality) are also applied. SQA, SCM, documentation, and measurement are conducted in streamlined manner.
- Strict
  In this case a full process is implemented and all umbrella activities are applied. The work products generated in this case are robust.
- Quick Reaction
  This approach is taken in case of an emergency. In this case only task essential for maintaining good quality are applied. After the task has been accomplished, documents are updated by back-filling.

The next question is how to decide about the degree of rigor. For this purpose an adaptation criterion has been developed. The following parameters are considered before a decision is made:

- Size of the project
- Number of potential users
- Mission criticality
- Application longevity
- Stability of requirements
- Ease of customer/developer communication
- Size of the project
- Number of potential users
- Mission criticality
- Application longevity
- Stability of requirements
- Ease of customer/developer communication

These parameters are used to calculate what is known as **Task Set Selector (TSS)** which is then used to determine the degree of rigor. It is computed as follows:

Each adaptation criteria is assigned a grade, weight, and entry point multiplier.

1. A grade value of between 1 and 5 is assigned to each adaptation criteria as appropriate.
2. The default value of the weighting factor (as given in the table below) is reviewed and modify if needed. The ranges should be between 0.8 to 1.2.

Adaptation Criteria	Grade	Weight	Entry Point Multiplier					Product
			Conc.	New Dev.	Enhan.	Maint.	Reeng.	
Size of product		1.2	0	1	1	1	1	
Number of Users		1.1	0	1	1	1	1	
Business Criticality		1.1	0	1	1	1	1	
Longevity		0.9	0	1	1	0	0	
Stability of requirements		1.2	0	1	1	1	1	
Ease of communication		0.9	0	1	1	1	1	
Maturity of technology		0.9	1	1	0	0	1	
Performance Constraints		0.8	0	1	1	0	1	
Embedded / non-embedded		1.2	1	1	1	0	1	
Project Staffing		1.0	1	1	1	1	1	
Interoperability		1.1	0	1	1	1	1	
Reengineering factors		1.2	0	0	0	0	1	
Task set selector								

3. Multiply the grade entered by the weight and by entry point multiplier for the type of project undertaken. The entry point multiplier takes a value of 0 or 1 and indicates the relevance of the adaptation criterion to the project type.

Product = grade × weight × entry point multiplier

The TSS is then computed as the average of all the entries in the product column. It is then used to determine the degree of rigor required as shown in the table below.

TSS	**Degree of Rigor**
< 1.2	Casual
1.0 – 3.0	Structured
> 2.4	Strict

It may be noted that there is overlap in TSS values. This overlap is intentional and it depicts that sharp boundaries are impossible to define when making task set selection. In the final analysis, the TSS value, past experience, and common sense must all be factored into the choice of the task set for a project.

This concept is demonstrated with the help of following example. In this case TSS value is computed for a new development project.

Adaptation Criteria	Grade	Weight	Entry point multiplier	Product
			New Development	
Size of product	2	1.2	1	2.4
Number of Users	3	1.1	1	3.3
Business Criticality	4	1.1	1	4.4
Longevity	3	0.9	1	2.7
Stability of requirements	2	1.2	1	2.4
Ease of communication	2	0.9	1	1.8
Maturity of technology	2	0.9	1	1.8
Performance Constraints	3	0.8	1	2.4
Embedded / non-embedded	3	1.2	1	3.6
Project Staffing	2	1.0	1	2.0
Interoperability	4	1.1	1	4.4
Reengineering factors	0	1.2	0	0
TSS				2.6

Since the value of TSS is 2.6, the degree of rigor could be structured or strict. The project manager now needs to use his experience to determine which of the two approaches should be taken.

TSS	**Degree of Rigor**
< 1.2	Casual
1.0 – 3.0	Structured
> 2.4	Strict

Example – SE Tasks for a Concept Development Project

Lecture No. 23

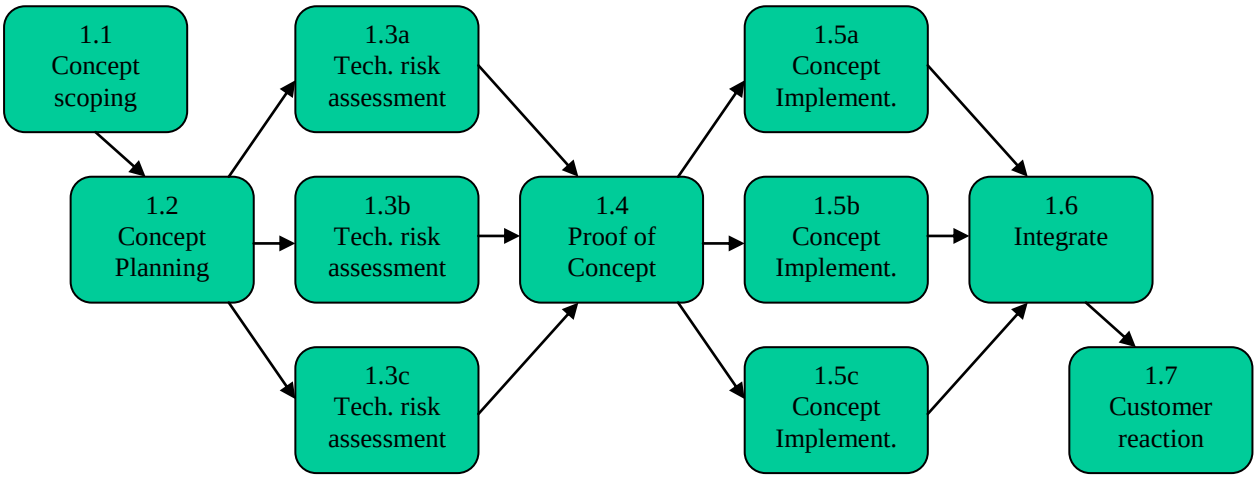
Task Network

Let us now apply the principles studied above to develop a plan for a concept development project. In this case the potential for new technology or new business idea is explored. Since it is a concept development project, the applicability is not certain but it appears to be useful and hence needs to be explored. Major tasks include:

- Concept scoping
- Preliminary concept planning
- Technology risk assessment
- Proof of concept
- Concept implementation
- Customer reaction to concept

Defining a Task Network

Once the tasks have been identified, we need to develop a task network to determine the sequence in which these activities need to be performed. This will ultimately lead to the time required to complete the project (to be discussed later). The following diagram shows the task network for the above project.



Lecture No. 24

Scheduling

Once we have the task network, we are now ready to prepare a schedule for the project. For this we use two techniques known as:

- Program evaluation and review techniques (PERT)
- Critical Path Method (CPM)

These are quantitative tools that allow the software planner to determine the critical path – the chain of tasks that determines the duration of the project and establish most likely time estimates for individual tasks by applying statistical models. They also help the planner to calculate boundary times that define a time window for a particular task.

The boundary time defines the following parameters for a project:

- The earliest time that a task can begin when all preceding tasks are completed in the shortest possible time
- The latest time for task initiation before the minimum project completion time is delayed
- The earliest finish
- The latest finish
- The total float – the amount of surplus time or leeway allowed in scheduling tasks so that the network critical path is maintained on schedule

In order to use the PERT and CPM, the following is required:

- A decomposition of product function
- A selection of appropriate process model and task set
- Decomposition of tasks – also known as the work breakdown structure (WBS)
- Estimation of effort
- Interdependencies

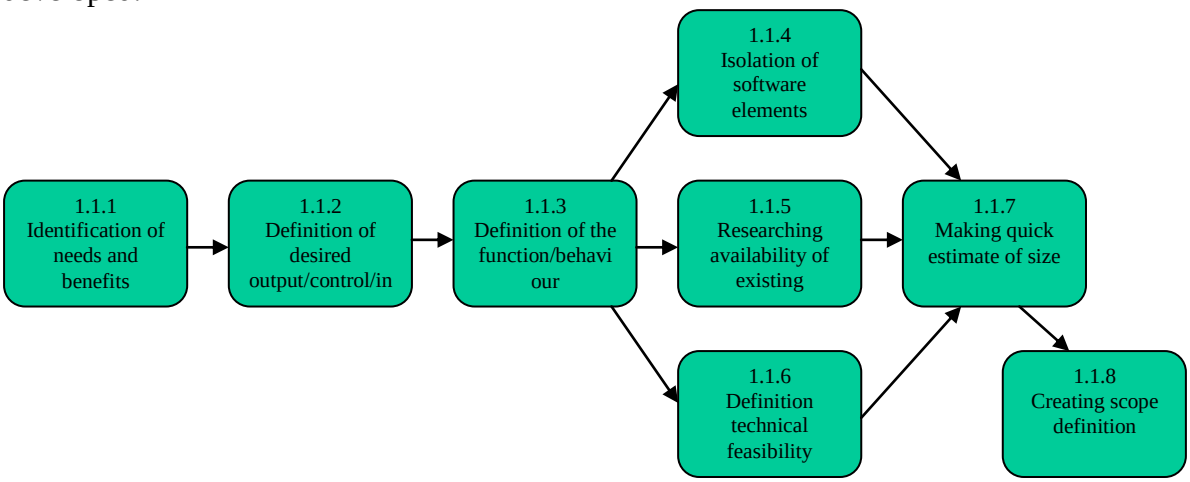
Timeline Chart

To develop the schedule for a project, time required for each activity in the Task Network is estimated. This analysis and decomposition leads to the development of a Timeline or Gantt Chart for the project which portrays the schedule for the project. As an example, let us assume that Concept Scoping (the first task in the above list) is further subdivided into the following sub-tasks with the associated estimated time requirements:

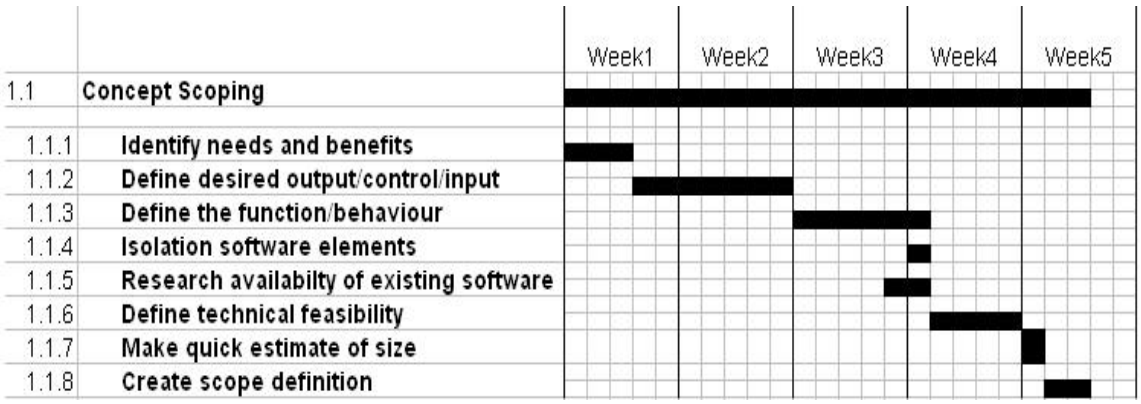
1. Identification of needs and benefits (3 days)
2. Definition of desired output/control/input (7 days)
3. Definition of the function/behaviour (6 days)
4. Isolation of software elements (1 day)
5. Researching availability of existing software (2 days)
6. Definition technical feasibility (4 days)

7. Making quick estimate of size (1 day)
8. Creating scope definition (2 days)

We also assume that the following task network for this was developed.



This is now converted in the following schedule in the form of a Gantt Chart. Note that, the concept of boundary time allows us to schedule Task Numbers 1.1.4 and 1.1.5 anywhere along Task Number 1.1.3. The actual time is determined by the project manager is based upon the availability of resources and other constraints. Each task is further subdivided in sub-tasks in the same manner until the schedule for the complete project is determined.



Tracking a Schedule

Lecture No. 25

Project Tracking

A schedule is meaningless if it is not followed and tracked. Tasks and milestones defined in a project schedule must be tracked and controlled as project proceeds. Tracking methods include:

- Periodic project status meetings
- Evaluating the results of all reviews
- Determine whether project milestones have been accomplished by the scheduled date
- Comparing actual start date to planned start date
- Informal meetings with the practitioners
- Using earned value analysis
- Error tracking

The last two techniques are discussed in further detail in the following paragraphs:

Earned Value Analysis

Earned Value Analysis or EVA is a quantitative technique for assessing the progress of a project. The earned value system provides a common value scale for every software task, regardless of the type of work being performed. The total hours to do the whole project are estimated, and every task is given an earned value based on the estimated percentage of the total. In order to do the EVA, the budgeted cost of work schedule (BCWS) is determined as follows:

Let

$BCWS_i$ = effort (person-days etc) for task*i*

BCWS is then the Progress so far – add all BCWS*i* so far.

Now

BAC = budget at completion = $\sum$ BCWS*i*

Now if BCWP is the Budgeted Cost of Work Perform, then

Schedule performance index	SPI = BCWP/BCWS
Schedule variance	SV = BCWP – BCWS

SPI close to 1 indicates efficient execution.

Similarly

Percent scheduled for completion = BCWS/BAC
Percent complete = BCWP/BAC
Actual cost work performed ACWP
Cost performance index CPI = BCWP/ACWP
Cost variance CV = BCWP – ACWP

Now, value of CPI close to 1 means project is within its defined budget.

Therefore, by using SPI and CPI we estimate how the project is progressing. If we have these values close to 1, it means that we have had good estimates and the project is under control.

© Virtual University of Pakistan

Lecture No. 26

Error Tracking

Error tracking can also be used to estimate the progress of the project. In this case we track errors in work products (requirement specifications, design documents, source code etc) to assess the status of a project. The process works as follows:

We collect error related metrics over many projects and determine our defect removal efficiency in the following manner:

Defect removal efficiency, DRE = E / (E+D), where

- E – errors found before shipment
- D – errors found during operation

It provides a strong indication of the effectiveness of the quality assurance activities.

Now let us assume that we have collected the following errors and defect data over the last 24 months:

- Errors per requirement specification page – E_{req}
- Error per component – design level – E_{design}
- Errors per component – code level – E_{code}
- DRE – requirement analysis
- DRE – architectural design
- DRE – coding

We now record the number of errors found during each SE step and calculate current values for E_{req}, E_{design}, and $E_{code.}$ These values are then compared to averages of past projects. If the current results vary more than 20% from average, there may be cause for concern and there is certainly cause for investigation.

Example

- E_{req} for the current project = 2.1
- Organizational average = 3.6
  - Two possibilities
    - The team has done an outstanding job
    - The team has been lax in its review approach
  - If the second scenario appears likely
    - Build additional design time

This can also be used to better target review and/or testing resources in the following manner:

- 120 components
- 32 exhibit E_{design} > 1.2 average
- Adjust code review resources accordingly

Time Boxing

Time-boxing is used in severe deadline pressure. It is a use incremental strategy where tasks associated with each increment are time-boxed in the following manner:

- Schedule for each task is adjusted by working backward from the delivery date.
- A box is put around each task
- When a task hits the boundary of the box, work stops and next task begins

The principle behind time-boxing is the 90-10 rule (similar to Pareto Principle) – rather than becoming stuck on the 10% of a task, the product proceeds towards the delivery date in 90% of the cases.

Lecture No. 27

Software Quality Assurance

Quality cannot be assured without first understanding its nature and characteristics. So the first question one has to ask is: what is quality?

Software quality is defined as conformance to explicitly stated functional and non-functional requirements, explicitly documented development standards, and implicit characteristics that are expected of all professionally developed software.

This definition emphasizes upon three important points:

- Software requirements are the foundation from which quality is measured. Lack of conformance is lack of quality
- Specified standards define a set of development criteria that guide the manner in which software is engineered. If the criteria are not followed, lack of quality will almost surely result.
- A set of implicit requirements often goes unmentioned (ease of use, good maintainability etc.)

Another very important question is: Do you need to worry about it after the code has been generated? In fact, SQA is an umbrella activity that is applied throughout the software process.

Also, do we care about internal quality or the external quality? And finally, is there a relationship between internal and external qualities? That is, does internal quality translate in external quality?

In the literature, quality has been defined through in many different manners. One group believes that the quality has measurable characteristic such as cyclomatic complexity, cohesion, and coupling.
We can then talk about quality from different aspects. Quality of design tries to determine the quality of design related documents including requirements, specifications, and design. Quality of conformance looks at the implementation and if it follows the design then the resulting system meets its goals then conformance quality is high.
Are there any other issues that need to be considered? Glass defines quality as a measure of user satisfaction which is defined by

compliant product + good quality + delivery within budget and schedule

DeMarco defines product quality as a function of how much it changes the world for the better.

So, there are many different way to look at the quality.

Quality Assurance
Goal of quality assurance is to provide the management with the necessary data to be informed about product quality. It consists of auditing and reporting functions of

management. If data provided through QA identifies problems, the management deploys the necessary resources to fix it and hence achieves desired quality control.

Cost of quality

A very significant question is: does quality assurance add any value. That is, is worth spending a lot of money in quality assurance practices? In order to understand the impact of quality assurance practices, we have to understand the cost of quality (or lack thereof) in a system.
Quality has a direct and indirect cost in the form of cost of prevention, appraisal, and failure.
If we try to prevent problems, obviously we will have to incur cost. This cost includes:

- Quality planning
- Formal technical reviews
- Test equipment
- Training

We will discuss these in more detail in the later sections.
The cost of appraisal includes activities to gain insight into the product condition. It involves in-process and inter-process inspection and testing.
And finally, failure cost. Failure cost has two components: internal failure cost and external failure cost. Internal failure cost requires rework, repair, and failure mode analysis. On the other hand, external failure cost involves cost for complaint resolution, product return and replacement, help-line support, warranty work, and law suits.
It is trivial to see that cost increases as we go from prevention to detection to internal failure to external failure. This is demonstrated with the help of the following example:
Let us assume that a total of 7053 hours were spent inspecting 200,000 lines of code with the result that 3112 potential defects were prevented. Assuming a programmer cost of $40 per hour, the total cost of preventing 3112 defects was $382,120, or roughly $91 per defect.
Let us now compare these numbers to the cost of defect removal once the product has been shipped to the customer. Suppose that there had been no inspections, and the programmers had been extra careful and only one defect one 1000 lines escaped into the product shipment. That would mean that 200 defects would still have to be fixed in the field. As an estimated cost of $25000 per fix, the cost would be $5 Million or approximately 18 times more expensive than the total cost of defect prevention

That means, quality translates to cost savings and an improved bottom line.

SQA Activities

There are two different groups involved in SQA related activities:

- Software engineers who do the technical work
- SQA group who is responsible for QA planning, oversight, record keeping, analysis, and reporting

Software engineers address quality by applying solid technical methods and measures, conducting formal and technical reviews, and performing well planned software testing.
The SQA group assists the software team in achieving a high quality product.

SQA Group Activities

An SQA plan is developed for the project during project planning and is reviewed by all stake holders. The plan includes the identification of:

- Evaluations to be performed
- Audits and reviewed to be performed
- Standards that are applicable to the project
- Procedures for error reporting and tracking
- Documents to be produced by the SQA group
- Amount of feedback provided to the software project team

The group participates in the development of the project's software process description. The software team selects the process and SQA group reviews the process description for compliance with the organizational policies, internal software standards, externally imposed standards, and other parts of the software project plan.
The SQA group also reviews software engineering activities to verify compliance with the defined software process. It identifies, documents, and tracks deviations from the process and verifies that the corrections have been made. In addition, it audits designated software work products to verify compliance with those defined as part of the software process. It, reviews selected work products, identifies, documents, and tracks deviations; verifies that corrections have been made; and reports the results of its work to the project manager.
The basis purpose is to ensure that deviations in software work and work products are documented and handled according to documented procedures. These deviations may be encountered in the project plan, process description, applicable standards, or technical work products. The group records any non-compliance and reports to senior management and non-compliant items are recorded and tracked until they are resolved.
Another very important role of the group is to coordinate the control and management of change and help to collect and analyze software metrics.

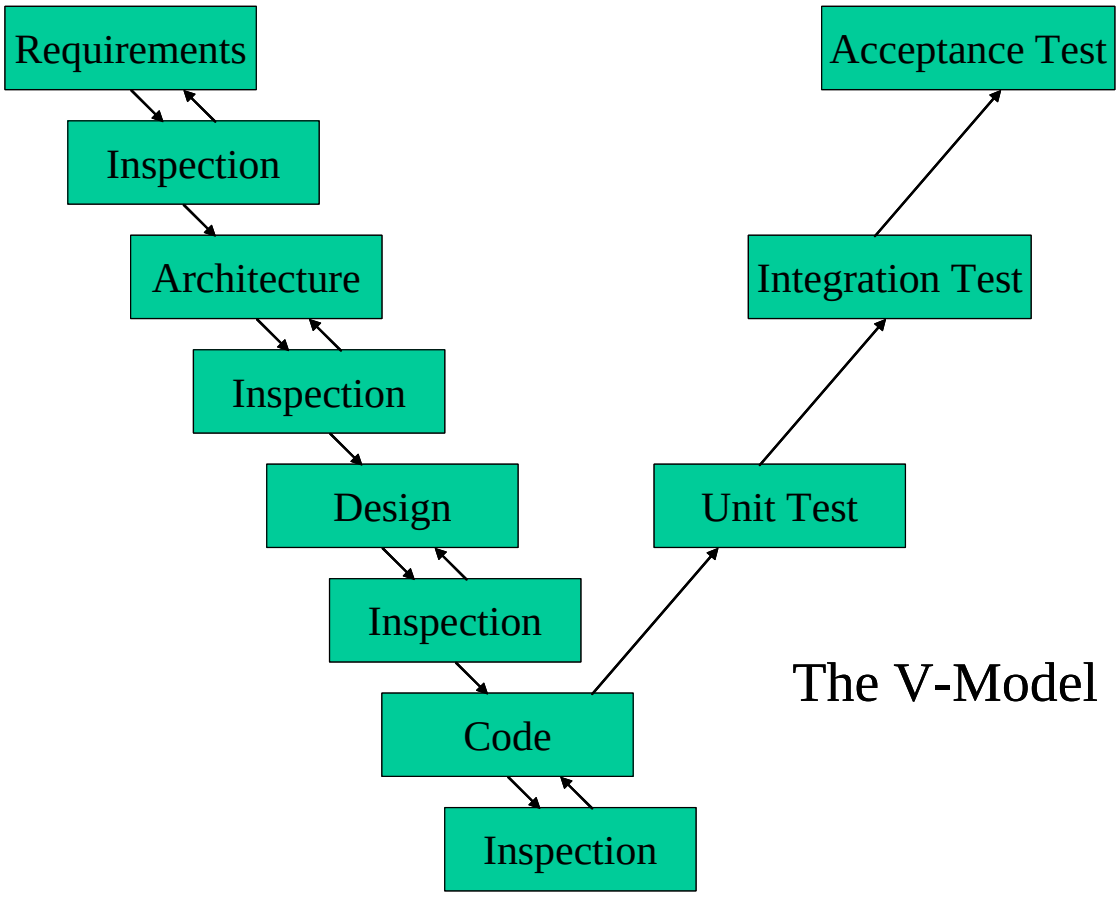
Quality Control

The next question that we need to ask is, once we have defined how to assess quality, how are we going to make sure that our processes deliver the product with the desired quality. That is, how are we going to control the quality of the product?
The basic principle of quality control is to control the variation as variation control is the heart of quality control. It includes resource and time estimation, test coverage, variation in number of bugs, and variation in support.
From one project to another we want to minimize the predicted resources needed to complete a project and calendar time. This involves a series of inspection, reviews, and tests and includes feedback loop. So quality control is a combination of measurement and feedback and combination of automated tools and manual interaction.

Lecture No. 28

Software Reviews

Software reviews are the filter for the software engineering process. They re applied at various different points and serve to uncover errors that can be removed and help to purify the software engineering activities.
In this context it is useful to look at the "V-model" of software development. This model emphasizes that SQA is a function performed at all stages of software development life cycle. At the initial stages (requirement, architecture, design, code), it is achieved through activities known as Formal Technical Reviews or FTR. At the later stages (integration and acceptance), testing comes into picture.



The V-Model

Importance of reviews

Technical work needs reviewing for the same reason that pencils needs erasers: To err is human. The second reason that we need technical reviews is although that people are good at catching errors, large class of errors escape the originator more easily than they escape anyone else.

Freedman defines a review – any review – as a way of using the diversity of a group of people to:

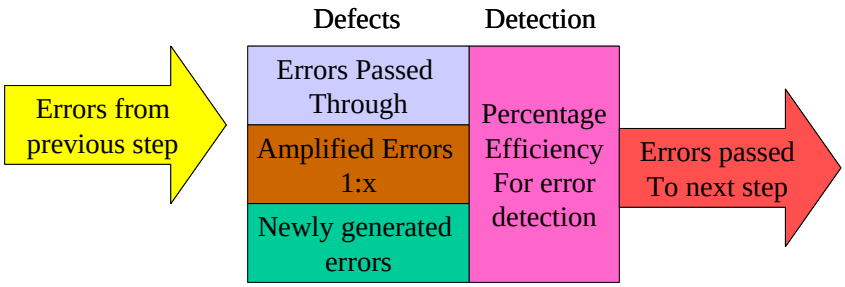
- Point out needed improvements in the product of a single person or team
- Confirm those parts of a product in which improvement is either not desired or no needed
- Achieve technical work of more uniform, or at least more predictable, quality than can be achieved without reviews, in order to make technical work more manageable.

Reviews help the development team in improving the defect removal efficiency and hence play an important role in the development of a high-quality product.

Types of Reviews

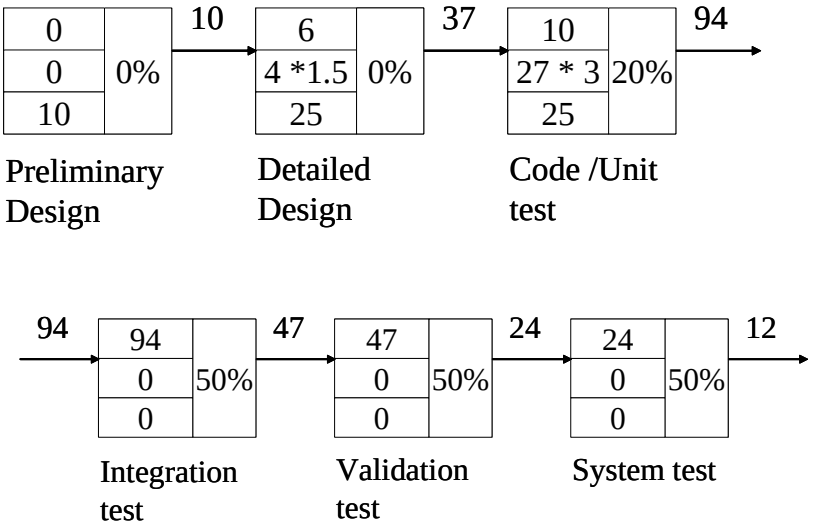
There are many types of reviews. In general they can be categorized into two main categories namely informal and formal technical reviews. Formal Technical reviews are sometimes called as walkthroughs or inspections. They are the most effective filter from QA standpoint. To understand the significance of these reviews, let us look at the defect amplification model shown below.

This model depicts that each development step inherits certain errors from the previous step. Some of these errors are just passed through to the next step while some are worked on and hence are amplified with a ratio of 1:x. In addition, each step may also generate some new errors. If each step has some mechanism for error detection, some of these errors may be detected and removed and the rest are passed on to the next step.

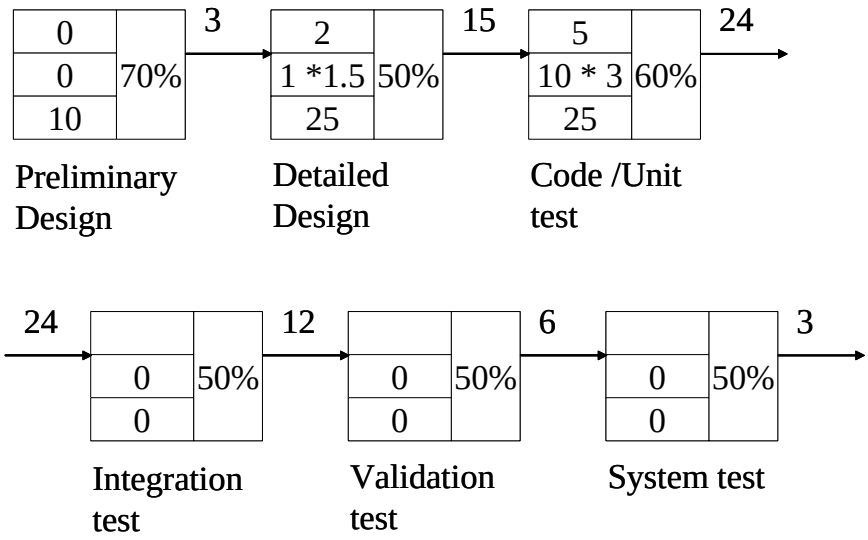


Let us now assume that we do not have any SQA related activities for the first two stages and we are only using testing for detection of any defects. Let us assume that the

Preliminary design generated 10 defects which were passed on to Detailed design. At that phase, 6 defects were passed on to the next stages and 4 were amplified at a ration of 1:1.5. In addition, there were 25 new defects introduced at this stage. Therefore, a total of 37 defects were passed on to the next stage as shown in the diagram. In the Code and Unite test stage, we start to test our system and assuming 20% defect removal efficiency of this stage, 94 defects (80% of (10 + 27 * 3 + 25)) are passed on to the next stage. This process continues and the system is delivered with 12 defects remaining in the product.



If FTR are used in the earlier stages, the quality of the end-product is much better as shown in the following diagram. Note that in this case we have code inspection in addition to unit testing at the third stage and the defect removal efficiency of that stage is 60%.



Lecture No. 29

Formal Technical Reviews

Formal Technical Reviews are conducted by software engineers. The primary objective is to find errors during the process so that they do not become defects after release of software as they uncover errors in function, logic design, or implementation. The idea is to have early discovery of errors so they do not propagate to the next step in the process. They also ensure that the software has been represented according to predefined standards and it is developed in a uniform manner. They make projects more manageable and help groom new resources as well as provide backup and continuity.

FTRs include walkthroughs, inspections, and other small group technical assessments of software.

Guidelines for walkthroughs

FTRs are usually conducted in a meeting that is successful only if it is properly planned, controlled, and attended. The producer informs the PM that the WP is ready and the review is needed. The review meeting consists of 3-5 people and advanced preparation is required. It is important that this preparation should not require more than 2 hours of work per person. It should focus on specific (and small) part of the overall software. For example, instead of the entire design, walkthroughs are conducted for each component, or small group of components. By narrowing focus, FTR has a high probability of uncovering errors.

It is important to remember that the focus is on a work product for which the producer of the WP asks the project leader for review. Project leader informs the review leader. The review leader evaluates the WP for readiness and if satisfied generates copies of review material and distributes to reviewers for advanced preparation. The agenda is also prepared by the review leader.

Review Meetings

Review meeting is attended by the review leader, all reviewers, and the producer. One of the reviewer takes the roles of recorder. Producer walks through the product, explaining the material while other reviewers raise issues based upon their advanced preparation. When valid problems or errors are recorded, the recorder notes each one of them. At the end of the RM, all attendees of the meeting must decide whether to:

- Accept the product without further modification
- Reject the product due to severe errors
  - Major errors identified
  - Must review again after fixing
- Accept the product provisionally
  - Minor errors to be fixed
  - No further review
  -

Review Reporting and Record keeping

During the FTR the recorder notes all the issues. They are summarized at the end and a review issue list is prepared. A summary report is produced that includes:

- What is reviewed
- Who reviewed it
- What were the findings and conclusions

It then becomes part of project historical record.

The review issue list

It is sometimes very useful to have a proper review issue list. It has two objectives.

- Identify problem areas within the WP
- Action item checklist

It is important to establish a follow-up procedure to ensure that items on the issue list have been properly addressed.

Review Guidelines

It is essential to note that an uncontrolled review can be worse than no review. The basis principle is that the review should focus on the product and not the producer so that it does not become personal. Remember to be sensitive to personal egos. Errors should be pointed out gently and the tone should be loose and constructive.

This can be achieved by setting an agenda and maintaining it. In order to do so, the review team should:

- Avoid drift
- Limit debate and rebuttal
- Enunciate problem areas but don’t try to solve all problems
- Take written notes
- Limit the number of participants and insist upon advanced preparation
- Develop a checklist for each product that is likely to be reviewed
- Allocate resources and schedule time for FTRs
- Conduct meaningful training for all reviewers
- Review your early reviews
- Determine what approach works best for you

Lecture No. 30

Statistical Software Quality Assurance

Statistical SQA is a technique that measures the quality in a quantitative fashion. It implies that information about defects is collected and categorized and an attempt is made to trace each defect to underlying cause. It uses Pareto Principle to identify vital causes (80% of defects can be traced to 20% of causes) and moves to correct the problems that have caused the defects.

Example

Let us assume that information about defects is collected for one year and categorized as follows:

1. Incomplete or erroneous specifications (IES)
2. Misinterpretation of customer communication (MCC)
3. Intentional deviation from specifications (IDS)
4. Violation of programming standards (VPS)
5. Error in data representation (EDR)
6. Inconsistent component interface (ICI)
7. Error in digital logic (EDL)
8. Incomplete or erroneous testing (IET)
9. Inaccurate or incomplete documentation (IID)
10. Error in programming language translation of design (PLT)
11. Ambiguous or inconsistent HCI (HCI)
12. Miscellaneous (MIS)

The following data is collected for these categories

Error Category	**Serious**	**Moderate**	**Minor**	**Sub Total**
IES	34	68	103	205
MCC	12	68	76	156
IDS	1	24	23	48
VPS	0	15	10	25
EDR	26	68	36	130
ICI	9	18	31	58
EDL	14	12	19	45
IET	12	35	48	95
IID	2	20	14	36
PLT	15	19	26	60
HCI	3	17	8	28
MIS	0	15	41	56

Total	128	379	435	942

We can easily see the following:

- IES, MCC, and EDR are the vital errors – cause 53% of all errors
- IES, EDR, PLT, and EDL are vital if only serious errors are considered

We now start corrective action focused on vital few. For example for EDR we review the data representation techniques to identify the possible improvement areas and adopt a use case tool for data modeling and perform stringent data design reviews.

Error Index (EI)

Another statistical technique known as Error Index (EI) is used to develop an overall indication of improvement in software quality. The EI is computed as follows:

Let

- E_i – the total number of errors uncovered during the ith step in the SE process
- S_i – number of serious errors
- M_i – number of moderate errors
- T_i – number of minor errors
- PS – product size at the ith step
- w_s, w_m, w_t – weighting factors for serious, moderate, and minor errors. Recommended values for these are 10, 3, 1 respectively.

At each step of the software process a *Phase Index* is computed as:

$$PI_i = w_s(S_i/E_i) + w_m(M_i/E_i) + w_t(T_i/E_i)$$

Now EI is computed as the cumulative effect on each $PI_i = \sum(I \times PI_i)/PS$

It is important to note that weighting errors encountered in the SE processes more heavily than those encountered earlier. As stated earlier, it can be used to develop an overall indication of improvement in software quality.

Software Reliability

Software reliability is another very important quality factor and is defined as probability of failure free operation of a computer program in a specified environment for a specified time. For example, a program X can be estimated to have a reliability of 0.96 over 8 elapsed hours.

Software reliability can be measured, directed, and estimated using historical and development data. The key to this measurement is the meaning of term failure. Failure is defined as non-conformance to software requirements. It can be graded in many different ways as shown below:

- From annoying to catastrophic
- Time to fix from minutes to months
- Ripples from fixing

It is also pertinent to understand the difference between hardware and software reliability. Hardware reliability is predicted on failure due to wear rather than failure due to design. In the case of software, there is no wear and tear. The reliability of software is determined by Mean time between failure (MTBF). MTBF is calculated as:

MTBF = MTTF + MTTR

Where MTTF is the Mean Time to Failure and MTTR is the Mean time required to Repair.

Arguably MTBF is far better than defects/kloc as each error does not have the same failure rate and the user is concerned with failure and not with total error count.

A related issue is the notion of availability. It is defined as the probability that a program is operating according to requirements at a given point in time. It can be calculated as

Availability = (MTTF/MTBF) x 100

and clearly depends upon MTTR.

Lecture No. 31

Software Safety

Software Safety is a software SQA activity that focuses on identification of potential hazards that may affect software negatively and cause an entire system to fail. Modeling and analysis process is conducted as part of software safety and hazards are identified and categorized by criticality and risk.

Example

Let us assume that the following hazards are associated with a computer-based cruise control for an automobile:

- Causes uncontrolled acceleration that cannot be stopped
- Does not respond to depression of brake pedal
- Does not engage when switch is activated
- Slowly loses or gains speed

Once system-level hazards are identified, analysis techniques are used to assign severity, and probability of occurrence. This technique is similar to risk analysis. To be effective, software must be analyzed in the context of the entire system. Analysis techniques such as fault tree analysis can be used to predict the chain of events that can cause hazards and the probability that each of these events will occur to create the chain. Once hazards are identified and analyzed, safety-related requirements can be specified for the software.

Reliability and safety are closely related. Software reliability uses statistical techniques to determine the likelihood that a software failure will occur. Occurrence of a software failure does not necessarily result in a hazard or mishap. On the other hand, software safety examines the ways in which failures result in conditions that can lead to a mishap.

Lecture No. 32

Poka-Yoke (Mistake-Proofing)

Poka-yoke is a QA technique developed by Shingo at Toyota in 1960's. Poka-yoke devices are mechanisms that lead to prevention of potential quality problem before it occurs or the rapid detection of quality problems if they are introduced.

- Examples:
  - Light on if the car door is not properly closed
  - Warning beep if the engine is turned-off when lights are on

Characteristic of a Poka-yoke device

- It is simple and cheap
- It is part of the process
- It is located near the process task where the mistake occurs

Example of Poka-yoke in software

Let us assume that a\A software Products Company sells application software to an international market. The pull-down menus and associated mnemonics provided with each application must reflect the local language. For example, the English language menu item for "Close" has the mnemonic "C" associated with it. When the application is sold to Urdu speaking users, the same menu item is "Band Karen" with mnemonic "ب". To implement the appropriate menu entry for each locale, a "localizer" translates the menus accordingly. The problem is to ensure that each menu entry conforms to appropriate standards and there are no conflicts, regardless of the language used.
We can consider a prevention device to write a program to generate mnemonics automatically, given a list of labels in each menu. It would prevent mistakes, but the problem of choosing a good mnemonic is difficult and the effort required would not be justified by the benefit gained. It is therefore not a good Poka-yoke device as it is not simple.
We now consider another prevention device. In this case we write a program to prevent the localizer from choosing mnemonics that did not meet the criteria. It would prevent mistakes but the benefit is minimal as incorrect mnemonics are easy enough to detect and correct after they occur. So once again this is not a good choice.
We now consider a detection device. In this case we write a program to verify that the chosen menu labels and mnemonics meet the criteria. We run this program to detect errors. Errors are sent back to localizer. Since this device requires a lot of back and forth movement, it violates the principle of co-location.
We now modify it to the following detection device. We write a program to verify that the chosen menu labels and mnemonics meet the criteria. The localizer would run this program to detect errors. It would give quick feedback and hence it is a good Poka-yoke device.

Lecture No. 33

Software Configuration Management (SCM)

You may recall that software configuration management (SCM) is one of the five KPA required for an organization to be at CMM level 2. That means, according to SEI, effective project management is not possible without having a proper SCM function in place.

The basic idea behind SCM is to manage and control change. As mentioned by Bersoff, no matter where you are in the system life cycle, the system will change, and the desire to change it will persist throughout the life cycle. It is therefore essential that we manage and control it in a fashion that this continuous change does not convert into chaos.

This has become more important in the context of contemporary software development as we are getting more and more complex software projects in terms of size, sophistication, and technology. In addition, these software systems are used by millions of users all over the world. These systems need multilingual and multi-platform support (hardware, software) and have to operate in a distributed environment. That means that a software system may come in many configuration flavors including desktop, standard, professional, and enterprise versions. There is a brutal competition out there and any complacency may result in losing a big market share. The huge maintenance frequency – corrective and adaptive – makes life even more difficult.

More complex development environment with shorter reaction time results in confusion and chaos!

Change Chaos

This frequent change, if not managed properly, results in chaos. First of all there would be problems of identification and tracking which would result in questions like the following:

- "This program worked yesterday. What happened?"
- "I fixed this error last week. Why is it back?"
- "Where are all my changes from last week?"
- "This seems like an obvious fix. Has it been tried before?"
- "Who is responsible for this change?"

Then there are problems of version selection. The typical problems faced are:

- "Has everything been compiled? Tested?"
- "How do I configure for test, with my updates and no others?"
- "How do I exclude this incomplete/faulty change?"
- "I can't reproduce the error in this copy!"
- "Exactly which fixes went into this configuration?"
- "Oh my God!. I need to merge 250 files!"
- Nobody knows which versions of programs are final
    - Where is the latest version?
    - Which version is the right one? I have so many
    - I have lost my latest changes
- Latest versions of code overwritten by old versions
- There was a minor problem, I fixed it but it is no longer working

- I can't figure-out the changes made to the previous version.
- I can't go back

Then there are software delivery problems.

- "Which configuration does this customer have?"
- "Did we deliver a consistent configuration?"
- "Did the customer modify the code?"
- "The customer skipped the previous two releases. What happens if we send him the new one?"
- Shipped the wrong version to the client.

This is not all. There may be more chaos in the following shapes and forms:

- The design document is out of sync with programs
- I don't know if all the changes that were suggested have been incorporated
- Which code was sent to testing?

SCM is a function that, if implemented, will reduce these problems to a minimal level.

Configuration management

As defined by CMM, the purpose of SCM is to establish and maintain the integrity or software products through the project's life cycle.
Configuration management is concerned with managing evolving software systems. It acknowledges that system change is a team activity and thus it aims to control the costs and effort involved in making changes to a system.
SCM involves the development and application of procedures and standards to manage an evolving software product and is part of a more general quality management process.

Baseline

When released to CM, software systems are called *baselines* and serve as the starting point for further development.A baseline is a software configuration management concept that helps us to control change without seriously impeding justifiable change. It is defined by IEEE as:

A specification or a product that has been formally reviewed and agreed upon, that thereafter serves as the basis for further development, and that can be changed only through formal change control procedures.

Software Configuration Item (SCI)

A Software Configuration Item is the information that is created as part of the software engineering process. Typical SCIs include requirement specifications, design specification, source code, test cases and recorded results, user guides and installation manuals, executable programs, and standards and procedures (for example C++ design guidelines).

Software Configuration Management Tasks

Software configuration management tasks include:

- Identification
- Version Control
- Change Control
- Configuration Auditing
- Reporting

Identification addresses how does an organization identify and manage the many existing versions of a program in a manner that will enable changes to be accommodated efficiently?

Version Control talks about how does an organization control changes before and after software is released to a customer? It is actually a combination of procedures and tools to manage different versions of the software configuration.

Clemm states that

> *Configuration management allows the user to specify alternative configurations of the software system through the selection of the appropriate versions. This is supported by associating with each software version, and then allowing configuration to be specified and constructed by describing the set of desired attributes.*

A version has many different attributes. In the simplest form a specific version number that is attached to each object and in the complex form it may have a string of Boolean variables (switches) that indicate specific types of functional changes that have been applied to the system.

The Change Control process addresses the important question of who has the responsibility for approving and ranking changes. Configuration Auditing deals with ensuring that the changes have been made properly and finally Reporting talks about the mechanism used to apprise others of changes that are made. Configuration Identification involves identification of a tool for SCM. Then a baseline is established and identified which is then used to identify configurable software items. At the minimum, all deliverables must be identified as configurable items. This includes design, software, test cases, tutorials, and user guides.

Lecture No. 34

Product Release Version Numbering System

Product release is the act of making a product available to its intended customers. After a product has had its first release, it enters a product release cycle. New versions of the product are made available that may fix defects or add features that were not in previous releases. These changes are categorized as updates or upgrades. An update fixes product defects. An upgrade enhances the product feature set and will include updates.

Release Numbering

Each individual product release is viewed as being in a unique state which is the total set of functionality possessed by the product release. Release numbering is a mechanism to identify the product's functionality state. Each release will have a different product state and hence will have a different release number. Although there is no industry standard, typically, a three field compound number of the format "X.Y.Z" is used. The different fields communicate functionality information about the product release.

The first digit, X, is used for the major release number which is used to identify a major increase in the product functionality. The major release number is usually incremented to indicate a significant change in the product functionality or a new product base-line.

The second digit, Y, stands for feature release number. The feature release number is iterated to identify when a set of product features have been added or significantly modified from their originally documented behaviour.

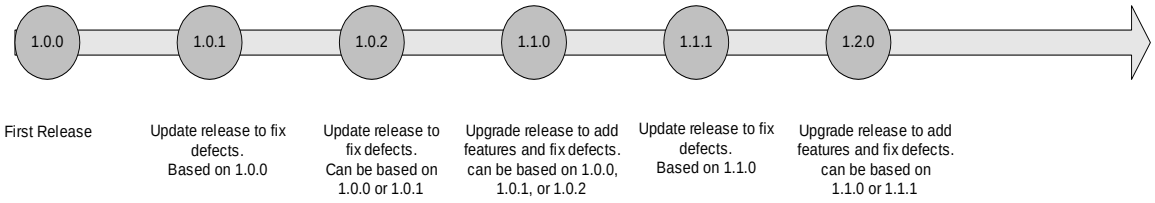
The third digit, Z, is called the defect repair number and is incremented when a set of defects is repaired. Defect repair/maintenance is considered to be any activity that supports the release functionality specification and it may a fix for some bugs or some maintenance to enhance the performance of the application.

Conventionally, a release number starts with a major number of one, followed by zero for its feature and maintenance numbers. This results in a release number 1.0.0. If the first new release that is needed is a defect repair release, the last digit would be iterated to one, resulting in 1.0.1. If two additional defect repair releases are needed, we would eventually have a release number of 1.0.3. Assume that an upgrade feature release is now needed. We will need to iterate the second field and will roll back the defect repair number to 0, resulting in a release number of 1.1.0. When we iterate the major release identifier, both the feature and defect numbers would be reset back to zero.

Internal Release Numbering

A special type of release is internal release. Internal releases are used by the development organization as a staging mechanism for functionality. The most common internal releases are the regular builds. A common way to number internal builds is to use the same release number that would be used for final release with some additional information added to it to identify the built. It is suggested that we add an extra (fourth) field to identify and keep track of internal builds.

The following diagram depicts the above mentioned version numbering system.



Change control

James Back points out the difficulties related to change control as follows:

Change control is vital. But the forces that make it necessary also make it annoying. We worry about change because a tiny perturbation in the code can cause a big failure in the product. But it can also fix a big failure or enable wonderful new capabilities. We worry about change because a single rogue developer could sink the project; yet brilliant ideas originate in the minds of those rogues, and a burdensome change control process could discourage them from doing creative work.

That is, like all engineering activities, change control is the name of a balancing act. Too much or too little change control creates different types of problems as uncontrolled change rapidly leads to chaos.

The change control process is elaborated in the following sub-section.

Lecture No. 35

Change Control Process

The first component of the change control process is the Change Control Authority (CCA) or a Change Control Board (CCB). A CCA or CCB includes people from both developer and client side.

Whenever a change is required, the CCB decides whether to allow this change to happen or deny it. If it is decided that a change is needed, an Engineering Change Order or ECO is generated. An ECO defines the change to be made, the constraints that must be respected, and the criteria for review and audit. The change control process thus involves the following steps.

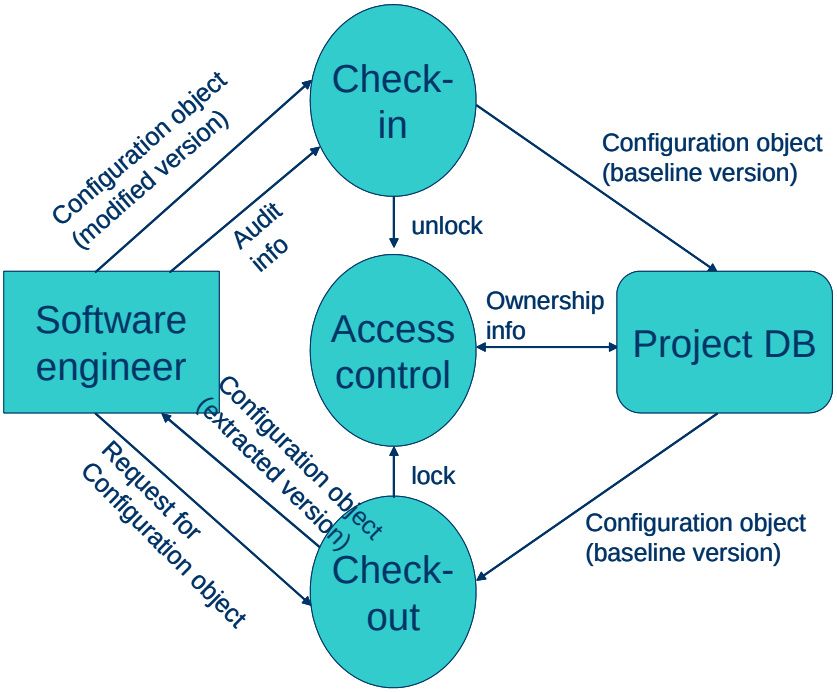
1) need for change is recognized
2) change request from user
3) developer evaluates
4) change report is generated
5) change control authority (CCA) decides
6) Either step 6a or 6b is performed. Steps numbers 7 to 17 are performed only if step 6b is performed.
   a)
      i) change request is denied
      ii) user is informed
      iii) no further action is taken.
   b) assign people to SCIs
7) check-out SCIs
8) make the change
9) review/audit the change
10) check-in SCIs
11) establish a “baseline” for testing
12) perform SQA and testing activities
13) check-in the changed SCIs
14) promote SCI for inclusion in next release
15) rebuild appropriate version
16) review/audit the change
17) include all changes in release

Thus, a change is incorporated in a controlled and strict manner.

Check-in and check-out

In SCM, the processes of Check-in and Check-out take a central stage. These are two important elements of change control and provide access and synchronization control. Access control manages who has the authority to check-out the object and synchronization control ensures that parallel changes by two different people do not overwrite one another.

Synchronization control implements a control on updates. When a copy is checked-out, other copies can be checked out for use only but they cannot be modified. In essence, it implements a single-writer multiple-readers protocol. This process is depicted in the following diagram.



Configuration Audit

Configuration audit ensures that a change has been properly implemented. It involves formal technical reviews and software configuration audit. Configuration audit assess a configuration object for characteristics that are generally not considered during audit. It is conducted by the SQA group. It looks into the following aspects of the change:

- Has the change specified in the ECO been made? Have any additional modifications been incorporated?
- Has a FTR been conducted to assess technical correctness?
- Has the software process been followed?
- Have the SE standards been properly applied?
- Has the change been highlighted in the SCI?
    - Change date and author
    - Have the attributes of the configuration object been updated?
- Have the SCM procedures for noting the change, recording it, and reporting it been followed?
- Have all related SCI's been properly updated?

An audit report is finally generated and any non-compliances are highlighted so that they may be corrected.

Configuration Status reporting (CSR)

Configuration Status Reporting (CSR) is also known as status accounting. It reports on the following items:

- What happened?
- Who did it?
- When did it happen?
- What else will be affected?

If it is not done then the organization faces the left hand not knowing what the right hand is doing syndrome. Without it, if a person responsible for the change leaves for whatever reason, it would be difficult to understand the whole scenario.

CSR reports are generated on regular basis. Each time an SCI is assigned a new identification, a CSR entry is made. Each time a change is approved by CCA, a CSR entry is made. Also, each time configuration audit is conducted, the results are reported as part of CSR task.

Lecture No. 36

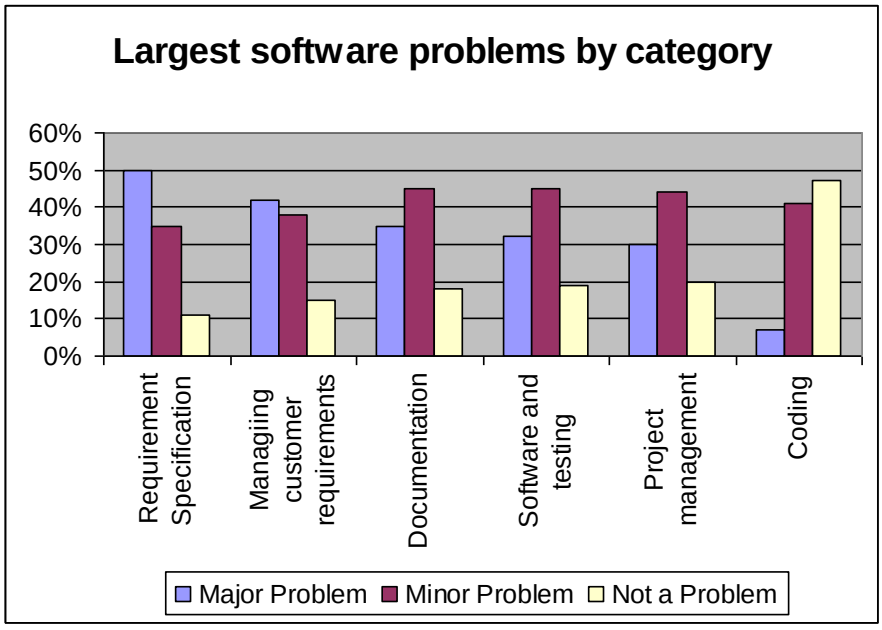
Requirement Management and CMM

CM standards

CM should always be based on a set of standards which are applied within an organisation. Standards should define how items are identified, how changes are controlled and how new versions are managed. Standards may be based on external CM standards (e.g. IEEE standard for CM ANSI/IEEE Std. No. 828-1983, 1042-1987, 1028-1988). Existing standards are based on a waterfall process model - new standards are needed for evolutionary development.

The Requirement Problem

The goal of software development is to develop quality software – on time and on budget – that meets customers' real needs. Project success depends on good requirement management. It may be recalled that requirement errors are the most common type of software development errors and the most costly to fix. It may also be recalled that requirement errors are listed as one of the roots causes of software project failure. According to a Standish Group report, lack of user input is responsible for13% of all project failures; incomplete requirements and specifications for 12% of all project failures; and changing requirements are responsible for 12% of all project failures.



The above graph shows the result of an industry survey. It is clear to see that requirement specifications are considered to be the most significant cause of major software problems by a majority of practitioners.

Requirement Management

Requirement management is also one of the 5 KPA defined at CMM level 2. Without having a proper requirement management function, chances of a project's success are slim.

Requirements Management KPA Goals statement says that:

1. The software requirements are controlled to establish a baseline for software engineering and management use.
2. Software plans, products, and activities are kept consistent with the software requirements.

Requirement Management is defined as a systematic approach to eliciting, organizing, and documenting the requirements of the system, and a process that establishes and maintains agreement between the customer and the project team on the changing requirements of the system.

It includes establishing and maintaining an agreement with the customer on the requirement for the software project. It involves

- Defining the requirement baseline
- Reviewing proposed requirement changes and evaluating the likely impact of each proposed change before deciding whether to approve it
- Incorporating approved requirement changes in the project in a controlled manner
- Keeping project plans current with the requirements
- Negotiating new commitments based on estimated impact on changed requirements
- Tracing individual requirements to their corresponding design, source code, and test cases
- Tracking requirement status and change activity throughout the project

Requirement Attributes

We need to tag requirements with certain attributes in order to manage them in an orderly fashion. Attributes are used to establish a context and background for each requirement. They go beyond the description of intended functionality. They can be used to filter, sort, or query to view selected subset of the requirements. A list of possible attributes is enumerated as below:

1. Requirement ID
2. Creation date
3. Created by
4. Last modified on
5. Last modified by
6. Version number
7. Status
8. Origin
9. Subsystem
10. Product Release
11. Priority

Requirement Status

The requirement status attribute is one of the most useful ones. It can be used to keep track of different requirements going through different phases. The possible status values are proposed, approved, implemented, verified, and deleted. These are elaborated in the following paragraphs.

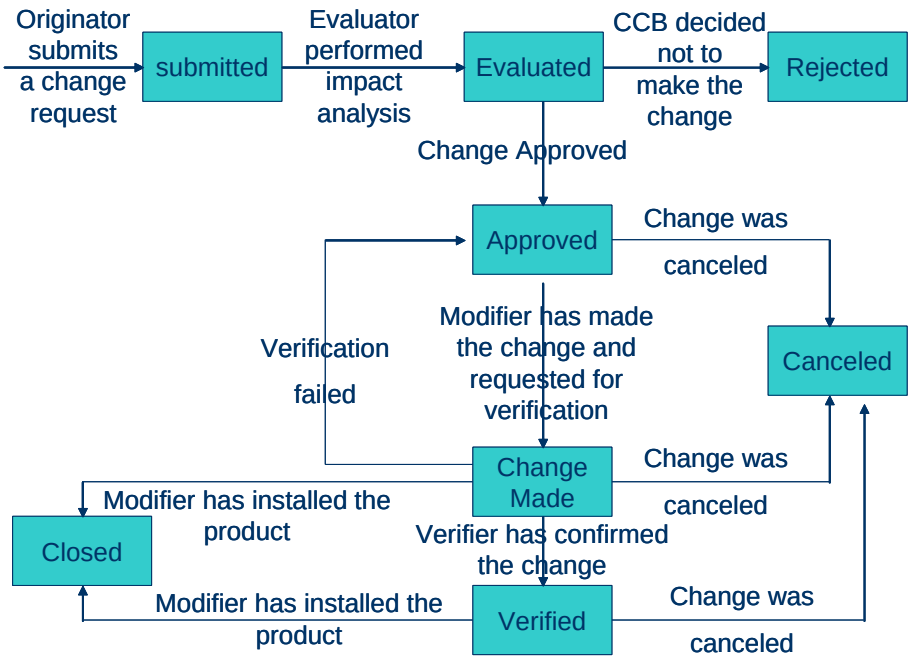
1. Proposed: The requirement has been requested by a source who has the authority to provide requirements.

2. Approved: The requirement has been analyzed, its impact on the rest of the project has been estimated, and it has been allocated to the baseline for a specific build number or product release. The software development group has committed to implement the requirement.
3. Implemented: The code that implements the requirement has been designed, written, and unit tested.

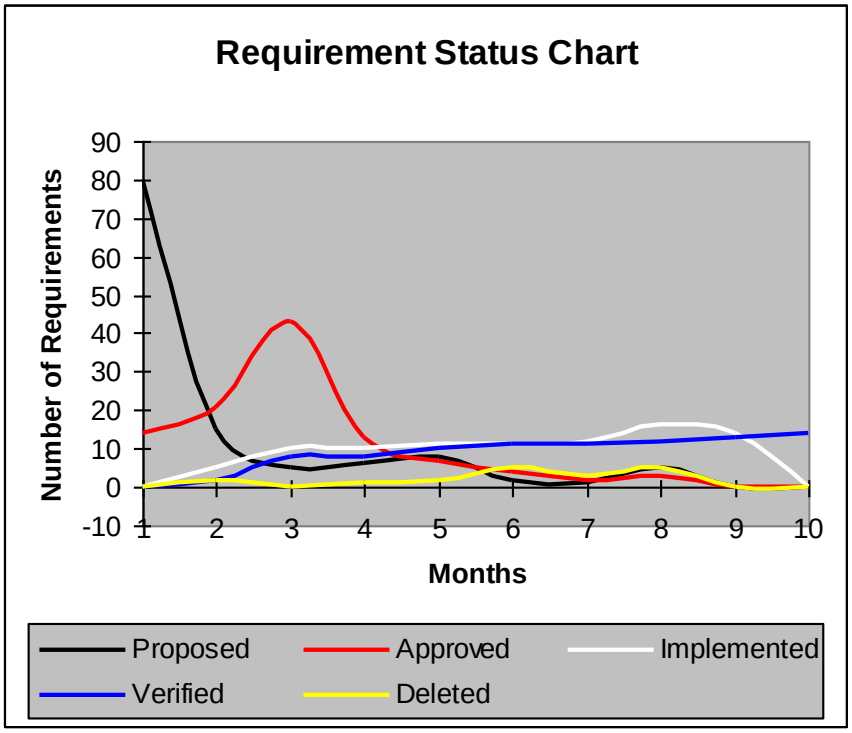
4. Verified: The implemented requirement has been verified through the selected approach, such as testing or inspection. The requirement has been traced to pertinent test cases. The requirement is now considered complete.
5. Deleted: A planned requirement has been deleted from the baseline. Include an explanation of why and by whom the decision was made to delete the requirement.

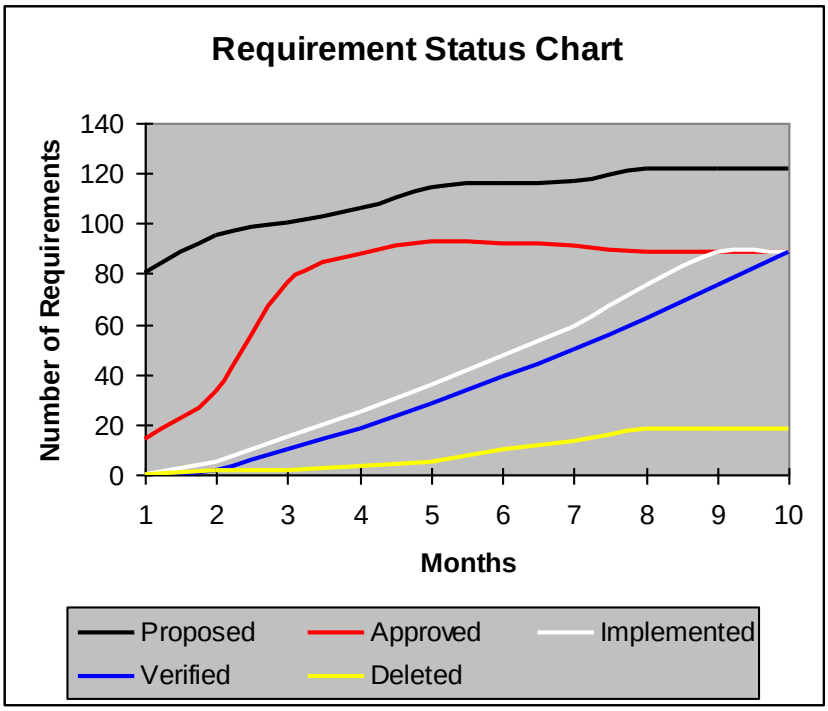
Change Request Status

As the requirement go through different phases, their status is updated accordingly. The following state transition diagram captures the sequence of requirement status changes.



The changes in the requirement status can be plotted as shown below to get an idea of the stability of the requirements and the progress of the project. It is easy to see that it is normal to have unstable requirements in the beginning but if they requirements stayed volatile till the end then the progress would be slow.





Managing Scope Creep

We must always remember that requirements will change, no matter what**.** That means we have to be able to manage changing requirements. Software organizations and professionals must learn to manage changing requirements. A major issue in requirements engineering is the rate at which requirements change once the requirements phase has "officially" ended. We therefore need to try to take it to a minimum level. For that we need to measure the change activity.

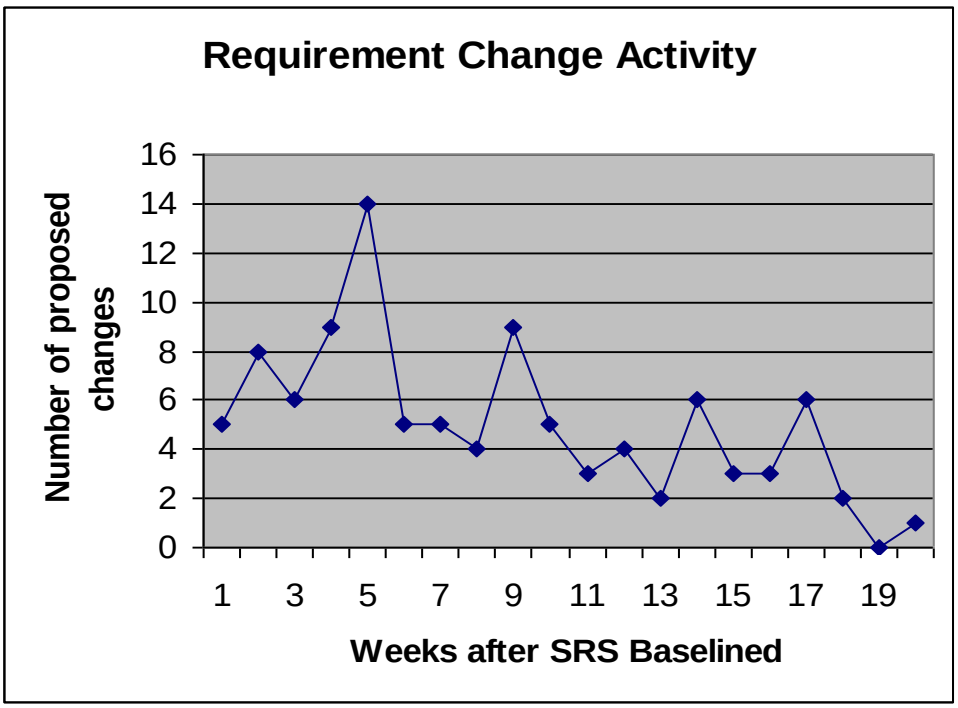
Lecture No. 37

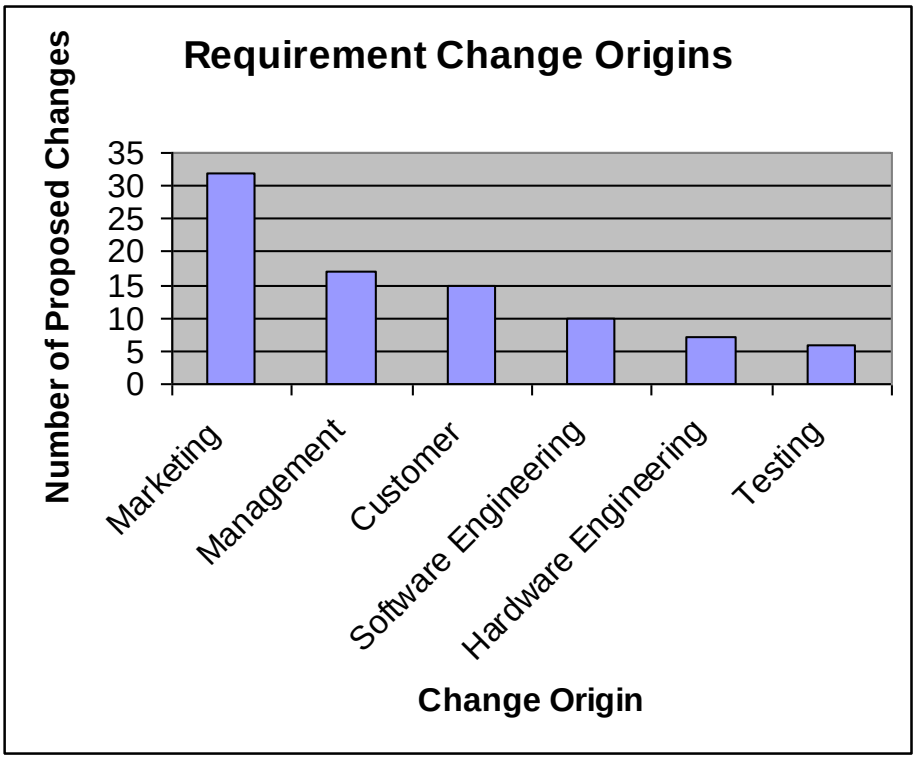
Measuring Change Activity

Measurement of change activity is a way to assess the stability of the requirements and to identify opportunities for process improvement. In this regards, the following could be measured

- The number of change requests received, open, and closed
- The cumulative number of changes made including added, deleted, and modified requirements
- The number of change requests that originated from each source
- The number of changes proposed and made in each requirement since it was base-lined
- The total effort devoted to handling changes

These can then be plotted as shown in the graphs below to get an idea of the stability of the systems. It is important to note that the sooner the requirements come to a stable state after establishing the baseline the better it is for the project. It is also useful to track the source of the requirement changes so that processes governing those areas causing more frequent changes may be strengthened in future projects.





Requirement Traceability

Requirement traceability is a very important consideration for requirement management. It is really hard to manage requirements that are not traceable.

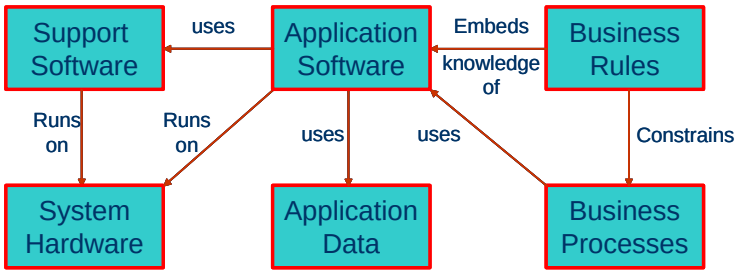
A Software Requirement Specification (SRS) is traced if the origin of its requirements is clear. That means that the SRS includes references to earlier supportive documents. An SRS is traceable if it written in a manner that facilitates the referencing of each individual requirement stated therein.

It is important to trace requirements both ways. That is from origin of a requirement to how it is implemented. This is a continuous process. It is also important that the rationale of requirements must also be traced. Traceability is important for the purposes of certification, change impact analysis, maintenance, project tracking, reengineering, reuse, risk reduction, and testing. That is it plays an important role in almost every aspect of the project and its life cycle.

Lecture No. 38

Legacy systems

A system is considered to be a legacy system if it has been in operation for many years. A legacy system has many components. These include business processes, business rules, application software, application data, support software, and system hardware. The relationship among these components is shown in the following diagram.



Maintaining Legacy System

Maintaining legacy system is expensive. It is often the case that different parts of the system have been implemented by different teams, lacking consistency. Part or all of the system may be implemented using an obsolete language. In most cases system documentation is inadequate and out of date. In some cases the only documentation is the source code. In some cases even the source code is not available.

Many years of maintenance have usually corrupted the system structure, making it increasingly difficult to understand. The data processed by the system may be maintained in different files which have incompatible structures. There may be data duplication and the documentation of the data itself may be out of date, inaccurate, and incomplete.

As far as the system hardware is concerned, the hardware platform may be outdated and is hard to maintain. In many cases, the legacy systems have been written for mainframe hardware which is no longer available, expensive to maintain, and not be compatible with current organizational IT purchasing policies.

Support software includes OS, database, and compiler etc. Like hardware, it may be obsolete and no longer supported by the vendors.

A time therefore comes when an organization has to make this decision whether to keep the old legacy system or to move it to new platform and environment. Moving it to new environment is known as legacy system migration.

Legacy migration risks

Legacy system migration however is not an easy task and there are a number of risks involved that need to be mitigated. First of all, there is rarely a complete specification of the system available. Therefore, there is no straight forward way of specifying the services provided by a legacy system. Thus, important business rules are often embedded

in the software and may not be documented elsewhere. Business processes and the way legacy systems operate are often intertwined. New software development may take several years.

New software development is itself risky as changes to one part of the system inevitably involve further changes to other components.

We therefore need to assess a legacy system before a decision for migration is made.

Legacy System Assessment

For each legacy system, there are four strategic options:

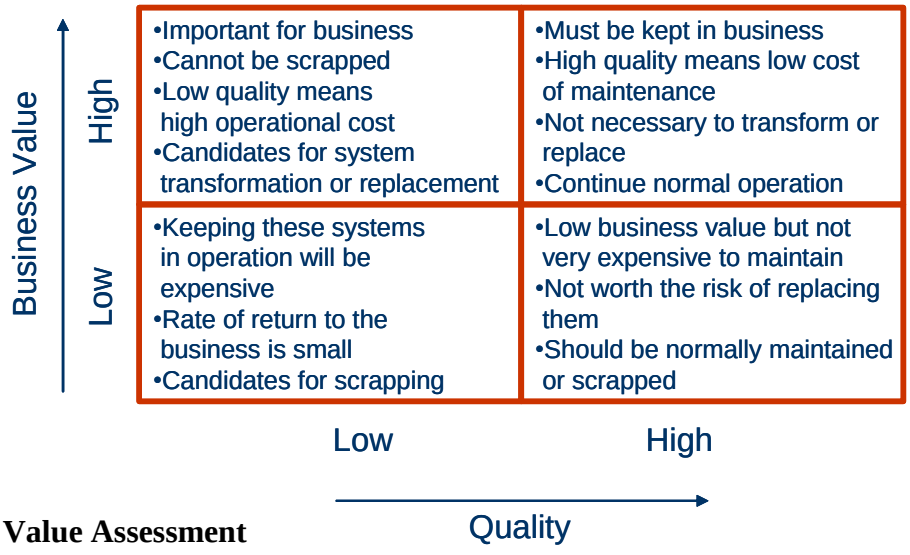
1. Scrap the system completely: This is the case when system is not making an effective contribution to business processes and business processes have changed significantly and the organization is no longer completely dependent upon the system.

2. Continue maintaining the system: This option is used when system is still required, it is stable, and requirements are not changing frequently

3. Transform the system in some way to improve its maintainability: this option is exercised when system quality has been degraded and regular changes to the system are required.

4. Replace the system with a new system: this path is taken when old system cannot continue in operation and off-the shelf alternative is available or system can be developed at a reasonable cost.

For these decisions, a legacy system can be assessed from two different perspectives – business value and quality. The following four quadrant assessment matrix can be used for this purpose.



Business Value Assessment

It is important to note that this is a subjective judgment and requires different business viewpoints. These view points include end-users, customers, line managers, IT managers, and senior managers.

End Users assess the system from the perspective of how effective do they find the system in supporting their business processes and how much of the system functionality is used.

The customers look at the system and ask is the use of the system transparent to customer or are their interaction constrained by the system, are they kept waiting because of the system, and do system errors have a direct impact on the customer.

From an IT Manager's perspective the following questions need to be asked: Are there difficulties in finding people to work on the system? Does the system consume resources which could be deployed more effectively on other systems?

Line Managers ask: do managers think that the system is effective in contributing to success of their unit? Is the cost of keeping the system in use justified? Is the data managed by the system critical for the functioning of the manager's unit?

Senior Managers look at the system from the angle that does the system and associated business process make an effective contribution to the business goal?

Lecture No. 39

Environment Assessment

The legacy system also needs to be assessed from an environment's perspective. This involves looking at the supplier, failure rate, age, performance, support requirements, maintenance cost, and interoperability.

These angles are elaborated in the following paragraph:

Supplier stability: Is the supplier still in existence? Is the supplier financially stable and likely to continue in existence? If the supplier is no longer in business, is the system maintained by someone else?

Failure rate: Does the hardware have a high rate of reported failure? Does the support software crash often and force system restarts?

Age: How old is the hardware and software?

Performance: Is the performance of the system adequate? Do performance problems have a significant effect on system users?

Support requirements: What local support is required by hardware and software? If there are high costs associated with this support, it may be worth considering system replacement?

Maintenance Cost: What are the costs of hardware maintenance and software licenses?
Interoperability: Are there problems interfacing the system with other systems? Can compilers etc be used with current versions of the operating system? Is system emulation required?

Application software assessment

The application software is assessed on the following parameters:

Understandability: How difficult is it to understand the software code of the current system? How complex are the control structures that are used?

Documentation: What system documentation is available? Is the documentation complete, consistent, and up-to-date?

Data: Is there an explicit data model for the system? To what extent is data duplicated in different files?

Programming Language: Are modern compilers available for the programming language? Is the language still used for new system development?

Test Data: Does test data for the system exist? Is there a record of regression tests carried out when new features have been added to the system?

Personnel skills: Are there people available who have the skills to maintain the system?

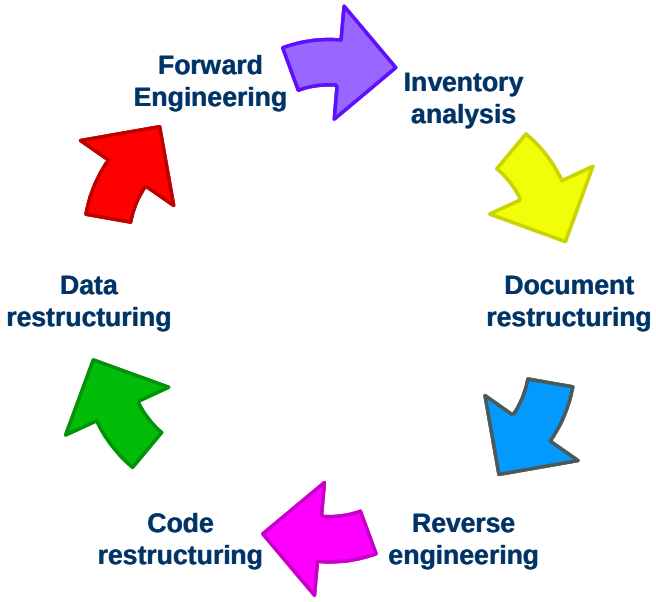
As migration is a very costly and risky business, the decision to migrate the system is made after assessing it from all these angles and it is determined that the time has come to migrate this system and it is worth spending the required amount of money and time for undertaking that effort.

Software Reengineering

Software solutions often automate the business by implementing business rules and business processes. In many cases, the software makes the business processes. As these rules and processes change, the software must also change. A time comes when these changes become very difficult to handle. So reengineering is re-implementing legacy systems to make them more maintainable. It is a long term activity.

Software Reengineering Process Model

The software reengineering is a non-trivial activity. Just like legacy migration, careful analysis must be carried out before a decision for reengineering is taken. The following process model can be used to reengineer a legacy system.



Inventory analysis

Inventory analysis is the first step in the reengineering process. At this stage, inventory of all applications is taken a note of their size, age, business criticality, and current maintainability is made. Inventory should be updated regularly as the status of the application can change as a function of time.

Document restructuring

The next step in the reengineering process is document restructuring. Weak documentation is a trademark of many legacy applications. Without proper documentation, the hidden rules, business processes, and data cannot be easily understood and reengineered.

In this regards, the following options are available:

1. Create documentation: Creating documentation from scratch is very time consuming. If program is relatively stable and is coming to the end of its useful life then just leave it as it is.
2. Update documentation: This option also needs a lot of resources. A good approach would be to update documentation when the code is modified.

Reverse engineering

Reverse engineering is the next step in the process. Reverse engineering for software is a process for analyzing a program in an effort to create a representation of the program at a higher level of abstraction than the source code. Reverse engineering is the process of design recovery. At this stage, documentation of the overall functionality of the system that is not there is created. The overall functionality of the entire system must be understood before more detailed analysis can be carried out.

Reverse engineering activities include:

- Reverse engineering to understand processing
- Reverse engineering to understand data
  - Internal data structures
  - Database structures
- Reverse engineering user interfaces

Program Restructuring

Program is restructured after the reverse engineering phase. In this case we modify source code and data in order to make it amenable to future changes. This includes code as well as data restructuring. Code restructuring requires redesign with same function with higher quality than original program and data restructuring involves restructuring the database or the database schema. It may also involve code restructuring.

Lecture No. 40

Forward Engineering

At the end forward engineering is carried out. It means incorporating the new business processes and rules in the system. Forward engineering requires application of SE principles, methods, and concepts to re-create an existing application. In most cases forward engineering does not simply create a modern equivalent of an older program, rather new user and technology requirements are integrated into the reengineering effort.

The Economics of Reengineering

As reengineering is a costly and risky undertaking, a cost benefit analysis for the reengineering effort must be carried out.

This analysis is carried out in the following manner.

Let

P1 **:** current annual maintenance cost for an application
P2 **:** current annual operation cost for an application
P3 **:** current annual business value of an application
P4 **:** predicted annual maintenance cost after reengineering
P5 **:** predicted annual operations cost after reengineering
P6 **:** predicted annual business value cost after reengineering
P7 **:** estimated reengineering cost
P8 **:** estimated reengineering calendar time
P9 **:** reengineering risk factor (1.0 is nominal)
L **:** expected life of the system

Now the cost of maintenance is calculated as:

$$C_{maintenance} = [P3 – (P1 + P2)] \times L$$

Cost of reengineering would then be given by the formula:

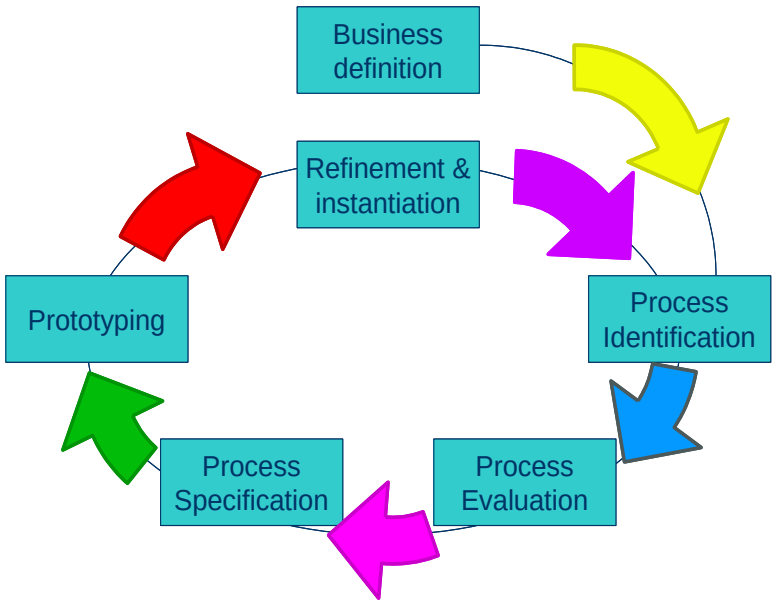
$$C_{reengineering} = [P6 – (P4 + P5) \times (L – P8) – (P7 \times P9)]$$

Lecture No. 41

Business Process Reengineering

A concept similar to software reengineering is of business process reengineering (BPR). A business process is "a set of logically related tasks performed to achieve a defined business outcome". It is the way certain business is conducted. Purchasing services and supplies, hiring new employees, paying suppliers are examples of business processes.

For BPR the following process model may be used.



As obvious from the diagram, it starts with the business definition where business goals are identified. The key drivers could be cost reduction, time reduction, quality improvement, and personnel development and empowerment. It may be defined at the business level or for a specific component of the business.

The next step is process identification. At this time processes that are critical to achieving the goals are identified and are ranked by importance, and need for change.

The short listed processes are then evaluated. Existing process is analyzed and measured and process tasks are identified. The cost and time consumed is measured as well as the quality and performance problems are identified.

Then, process specification and design is carried out. Use cases are prepared for each process to be redesigned and a new set of tasks are designed for the processes and then they are prototyped.

A redesigned business process must be prototyped before it is fully integrated into the business.

Based on the feedback the business process is refined.

Lecture No. 42

Software Refactoring

Software refactoring is the process of changing a software system such that the external behavior of the system does not change while the internal structure of the system is improved. This is sometimes called “Improving the design after it has been written”.

Fowler defines refactoring as A change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior. It is achieved by applying a series of refactoring without changing its observable behavior.

A (Very) Simple Example

Let us consider a very simple example. The refactoring involved in this case is known as “Consolidate Duplicate Conditional Fragments”. As the name suggests, this refactoring lets the programmer improve the quality of the code by grouping together the duplicate code, resulting in less maintenance afterwards.

```
if (isSpecialDeal()) {
   total = price * 0.95;
   send ();
}
else {
   total = price * 0.98;
   send ();
}
```

In this case, *send* is being called from different places. We can consolidate as follows:

```
if (isSpecialDeal())
   total = price * 0.95;
else
   total = price * 0.98;

send ();
```

It can further be improved if we calculate total outside the if statement as shown below.

```
if (isSpecialDeal())
   factor = 0.95;
else
   factor = 0.98;

total = price * factor;
send ();
```

Although this is a trivial example, it nevertheless is useful and teaches us how can be consolidate code by grouping together pieces of code from different segments.

Refactoring: Where to Start?

The first question that we have to ask ourselves is: how do you identify code that needs to be refactored? Fowler et al has devised a heuristic based approach to this end known as "Bad Smells" in Code. The philosophy is simple: ***"if it stinks, change it".***

Bad Smells in Code

They have identified many different types of "bad smells". These are briefly described in the following paragraphs:

- Duplicated Code
  - bad because if you modify one instance of duplicated code but not the others, you (may) have introduced a bug!
- Long Method
  - long methods are more difficult to understand; performance concerns with respect to lots of short methods are largely obsolete
- Large Class
  - Large classes try to do too much, which reduces cohesion
- Long Parameter List
  - hard to understand, can become inconsistent
- Divergent Change
  - Deals with cohesion; symptom: one type of change requires changing one subset of methods; another type of change requires changing another subset
- Shotgun Surgery
  - a change requires lots of little changes in a lot of different classes
- Feature Envy
  - A method requires lots of information from some other class (move it closer!)
- Data Clumps
  - attributes that clump together but are not part of the same class
- Primitive Obsession
  - characterized by a reluctance to use classes instead of primitive data types
- Switch Statements
  - Switch statements are often duplicated in code; they can typically be replaced by use of polymorphism (let OO do your selection for you!)
- Parallel Inheritance Hierarchies
  - Similar to Shotgun Surgery; each time I add a subclass to one hierarchy, I need to do it for all related hierarchies
- Lazy Class
  - A class that no longer "pays its way". e.g. may be a class that was downsized by refactoring, or represented planned functionality that did not pan out
- Speculative Generality
  - "Oh I think we need the ability to do this kind of thing someday"
- Temporary Field
  - An attribute of an object is only set in certain circumstances; but an object should need all of its attributes

- Message Chains
  - a client asks an object for another object and then asks that object for another object etc. Bad because client depends on the structure of the navigation
- Middle Man
  - If a class is delegating more than half of its responsibilities to another class, do you really need it?
- Inappropriate Intimacy
  - Pairs of classes that know too much about each other's private details
- Alternative Classes with Different Interfaces
  - Symptom: Two or more methods do the same thing but have different signature for what they do
- Incomplete Library Class
  - A framework class doesn't do everything you need
- Data Class
  - These are classes that have fields, getting and setting methods for the fields, and nothing else; they are data holders, but objects should be about data AND behavior
- Refused Bequest
  - A subclass ignores most of the functionality provided by its super-class
- Comments (!)
  - Comments are sometimes used to hide bad code
  - "…comments often are used as a deodorant" (!)

Breaking a Method

We have already seen example of duplicate code. We now look at another simple example of long method. Although, in this case, the code is not really long, it however demonstrates how longer segments of code can be broken into smaller and more manageable (may be more reusable as well) code segments.

The following code segment sorts an array of integers using "selection sort" algorithm.

```
for (i=0; i < N-1; i++) {
        min = i;
        for (j = i; j < N; j++)
                if (a[j] < a[min]) min = j;
        temp = a[i];
        a[i] = a[min];
        a[min] = temp;
}
```

We break it into smaller fragments by making smaller functions out of different steps in the algorithm as follows:

```
int minimum (int a[ ], int from, int to)
{
        int min = from;
```

```
        for (int i = from; i <= to; i++)
                if (a[i] < a[min]) min = i;
        return min;
}

void swap (int &x, int &y)
{
        int temp = x;
        x = y;
        y = temp;
}
```

The sort function now becomes simpler as shown below.

```
for (i=0; i < N-1; i++) {
        min = minimum (a, i, N-1);
        swap(a[i], a[min]);
}
```

It can be seen that it is now much easier to understand the code and hence is easier to maintain. At the same time we have got two separate reusable functions that can be used elsewhere in the code.

A slightly more involved example
(It has mostly been adapted from Fowler's introduction to refactoring which is freely available on the web)

Let us consider a simple program for a video store. It has three classes: Movie, Rental, and Customer. Program is told which movies a customer rented and for how long and it then calculates the charges and the Frequent renter points. Customer object can print a statement (in ASCII).

Here is the code for the classes. DomainObject is a general class that does a few standard things, such as hold a name.

```
public class DomainObject {

        public DomainObject (String name)    {
                _name = name;
        };

        public DomainObject ()       {};

        public String name ()        {
                return _name;
        };

        public String toString() {
                return _name;
```

```
        };

        protected String _name = "no name";
}
```

Movie represents the notion of a film. A video store might have several tapes in stock of the same movie

```
public class Movie extends DomainObject {
   public static final int  CHILDRENS = 2;
   public static final int  REGULAR = 0;
   public static final int  NEW_RELEASE = 1;


        private int _priceCode;

        public Movie(String name, int priceCode) {
                _name = name;
                _priceCode = priceCode;
        }

        public int priceCode() {
                return _priceCode;
        }

        public void persist() {
                Registrar.add ("Movies", this);
        }

        public static Movie get(String name) {
                return (Movie) Registrar.get ("Movies", name);
        }
}
```

The movie uses a class called a registrar (not shown) as a class to hold instances of movie. I also do this with other classes. I use the message persist to tell an object to save itself to the registrar. I can then retrieve the object, based on its name, with a get(String) method.

The tape class represents a physical tape.

```
class Tape extends DomainObject
   {
   public Movie movie() {
        return _movie;
   }
   public Tape(String serialNumber, Movie movie) {
        _serialNumber = serialNumber;
        _movie = movie;
   }
```

```
private String _serialNumber;
private Movie _movie;
}
```

The rental class represents a customer renting a movie.

```
class Rental extends DomainObject
{
public int daysRented() {
        return _daysRented;
}
public Tape tape() {
        return _tape;
}
private Tape _tape;
public Rental(Tape tape, int daysRented) {
        _tape = tape;
        _daysRented = daysRented;
}
private int _daysRented;
}
```

The customer class represents the customer. So far all the classes have been dumb encapsulated data. Customer holds all the behavior for producing a statement in its statement() method.

```
class Customer extends DomainObject
{
public Customer(String name) {
   _name = name;
}
public String statement() {
   double totalAmount = 0;
   int frequentRenterPoints = 0;
   Enumeration rentals = _rentals.elements();
   String result = "Rental Record for " + name() + "\n";
   while (rentals.hasMoreElements()) {
      double thisAmount = 0;
      Rental each = (Rental) rentals.nextElement();

      //determine amounts for each line
      switch (each.tape().movie().priceCode()) {
         case Movie.REGULAR:
            thisAmount += 2;
            if (each.daysRented() > 2)
               thisAmount += (each.daysRented() - 2) * 1.5;
            break;
         case Movie.NEW_RELEASE:
            thisAmount += each.daysRented() * 3;
            break;
```

```
            case Movie.CHILDRENS:
               thisAmount += 1.5;
               if (each.daysRented() > 3)
                  thisAmount += (each.daysRented() - 3) * 1.5;
               break;

         }
         totalAmount += thisAmount;

         // add frequent renter points
         frequentRenterPoints ++;
         // add bonus for a two day new release rental
         if ((each.tape().movie().priceCode() == Movie.NEW_RELEASE) &&
each.daysRented() > 1) frequentRenterPoints ++;

         //show figures for this rental
         result += "\t" + each.tape().movie().name()+ "\t" + String.valueOf(thisAmount) +
"\n";

      }
      //add footer lines
      result +=  "Amount owed is " + String.valueOf(totalAmount) + "\n";
      result += "You earned " + String.valueOf(frequentRenterPoints) + " frequent renter
points";
      return result;

   }
   public void addRental(Rental arg) {
         _rentals.addElement(arg);
   }
   public static Customer get(String name) {
         return (Customer) Registrar.get("Customers", name);
   }
   public void persist() {
         Registrar.add("Customers", this);
   }
   private Vector _rentals = new Vector();
   }
```

What are your impressions about the design of this program? I would describe as not well designed, and certainly not object-oriented. For a simple program is this, that does not really matter. There is nothing wrong with a quick and dirty *simple* program. But if we imagine this as a fragment of a more complex system, then I have some real problems with this program. That long statement routine in the Customer does far too much. Many of the things that it does should really be done by the other classes.

This is really brought out by a new requirement, just in from the users, they want a similar statement in html. As you look at the code you can see that it is impossible to reuse any of the behavior of the current statement() method for an htmlStatement(). Your only recourse is to write a whole new method that duplicates much of the behavior of

statement(). Now of course this is not too onerous. You can just copy the statement() method and make whatever changes you need. So the lack of design does not do too much to hamper the writing of htmlStatement(), (although it might be tricky to figure out exactly where to do the changes). But what happens when the charging rules change? You have to fix both statement() and htmlStatement(), and ensure the fixes are consistent. The problem from cut and pasting code comes when you have to change it later. Thus if you are writing a program that you donít expect to change, then cut and paste is fine. If the program is long lived and likely to change, then cut and paste is a menace.

But you still have to write the htmlStatement() program. You may feel that you should not touch the existing statement() method, after all it works fine. Remember the old engineering adage: "if it ainít broke, donít fix it". statement() may not be broke, but it does hurt. It is making your life more difficult to write the htmlStatement() method.

So this is where refactoring comes in. When you find you have to add a feature to a program, and the programís code is not structured in a convenient way to add the feature; then first refactor the program to make it easy to add the feature, then add the feature.

Extracting the Amount Calculation

The obvious first target of my attention is the overly long statement() method. When I look at a long method like that, I am looking to take a chunk of the code an *extract a method* from it.

Extracting a method is taking the chunk of code and making a method out of it. An obvious piece here is the switch statement (which I'm highlighting below).

```
public String statement() {
        double totalAmount = 0;
        int frequentRenterPoints = 0;
        Enumeration rentals = _rentals.elements();
        String result = "Rental Record for " + name() + "\n";
        while (rentals.hasMoreElements()) {
           double thisAmount = 0;
           Rental each = (Rental) rentals.nextElement();

           //determine amounts for each line
           switch (each.tape().movie().priceCode()) {
              case Movie.REGULAR:
                 thisAmount += 2;
                 if (each.daysRented() > 2)
                    thisAmount += (each.daysRented() - 2) * 1.5;
                 break;
              case Movie.NEW_RELEASE:
                 thisAmount += each.daysRented() * 3;
                 break;
              case Movie.CHILDRENS:
                 thisAmount += 1.5;
                 if (each.daysRented() > 3)
                    thisAmount += (each.daysRented() - 3) * 1.5;
```

```
                break;

            }
            totalAmount += thisAmount;

            // add frequent renter points
            frequentRenterPoints ++;
            // add bonus for a two day new release rental
            if ((each.tape().movie().priceCode() == Movie.NEW_RELEASE) &&
each.daysRented() > 1) frequentRenterPoints ++;

            //show figures for this rental
            result += "\t" + each.tape().movie().name()+ "\t" + String.valueOf(thisAmount) +
"\n";

        }
        //add footer lines
        result +=  "Amount owed is " + String.valueOf(totalAmount) + "\n";
        result += "You earned " + String.valueOf(frequentRenterPoints) + " frequent renter
points";
        return result;

    }
```

This looks like it would make a good chunk to extract into its own method. When we extract a method, we need to look in the fragment for any variables that are local in scope to the method we are looking at, that local variables and parameters. This segment of code uses two: each and thisAmount. Of these each is not modified by the code but thisAmount is modified. Any non-modified variable we can pass in as a parameter. Modified variables need more care. If there is only one we can return it. The temp is initialized to 0 each time round the loop, and not altered until the switch gets its hands on it. So we can just assign the result. The extraction looks like this.

```
public String statement() {
        double totalAmount = 0;
        int frequentRenterPoints = 0;
        Enumeration rentals = _rentals.elements();
        String result = "Rental Record for " + name() + "\n";
        while (rentals.hasMoreElements()) {
            double thisAmount = 0;
            Rental each = (Rental) rentals.nextElement();

            //determine amounts for each line
            thisAmount = amountOf(each);
            totalAmount += thisAmount;

            // add frequent renter points
            frequentRenterPoints ++;
            // add bonus for a two day new release rental
```

```
        if ((each.tape().movie().priceCode() == Movie.NEW_RELEASE) &&
each.daysRented() > 1) frequentRenterPoints ++;

        //show figures for this rental
        result += "\t" + each.tape().movie().name()+ "\t" + String.valueOf(thisAmount) +
"\n";

      }
      //add footer lines
      result +=  "Amount owed is " + String.valueOf(totalAmount) + "\n";
      result += "You earned " + String.valueOf(frequentRenterPoints) + " frequent renter
points";
      return result;

   }

   private int amountOf(Rental each) {
      int thisAmount = 0;
      switch (each.tape().movie().priceCode()) {
         case Movie.REGULAR:
            thisAmount += 2;
            if (each.daysRented() > 2)
               thisAmount += (each.daysRented() - 2) * 1.5;
            break;
         case Movie.NEW_RELEASE:
            thisAmount += each.daysRented() * 3;
            break;
         case Movie.CHILDRENS:
            thisAmount += 1.5;
            if (each.daysRented() > 3)
               thisAmount += (each.daysRented() - 3) * 1.5;
            break;

      }
      return thisAmount;
   }
```

When I did this the tests blew up. A couple of the test figures gave me the wrong answer. I was puzzled for a few seconds then realized what I had done. Foolishly I had made the return type of amountOf() int instead of double.

```
   private double amountOf(Rental each) {
      double thisAmount = 0;
      switch (each.tape().movie().priceCode()) {
         case Movie.REGULAR:
            thisAmount += 2;
            if (each.daysRented() > 2)
               thisAmount += (each.daysRented() - 2) * 1.5;
            break;
         case Movie.NEW_RELEASE:
```

```
            thisAmount += each.daysRented() * 3;
            break;
        case Movie.CHILDRENS:
            thisAmount += 1.5;
            if (each.daysRented() > 3)
                thisAmount += (each.daysRented() - 3) * 1.5;
            break;

    }
    return thisAmount;
}
```

Itís the kind of silly mistake that I often make, and it can be a pain to track down as Java converts ints to doubles without complaining (but merrily rounding). Fortunately it was easy to find in this case, because the change was so small. Here is the essence of the refactoring process illustrated by accident. Because each change is so small, any errors are very easy to find. You don't spend long debugging, even if you are as careless as I am.

This refactoring has taken a large method and broken it down into two much more manageable chunks. We can now consider the chunks a bit better. I don't like some of the variables names in amountOf() and this is a good place to change them.

```
private double amountOf(Rental aRental) {
    double result = 0;
    switch (aRental.tape().movie().priceCode()) {
        case Movie.REGULAR:
            result += 2;
            if (aRental.daysRented() > 2)
                result += (aRental.daysRented() - 2) * 1.5;
            break;
        case Movie.NEW_RELEASE:
            result += aRental.daysRented() * 3;
            break;
        case Movie.CHILDRENS:
            result += 1.5;
            if (aRental.daysRented() > 3)
                result += (aRental.daysRented() - 3) * 1.5;
            break;

    }
    return result;
}
```

Is that renaming worth the effort? Absolutely. Good code should communicate what it is doing clearly, and variable names are key to clear code. Never be afraid to change the names to things to improve clarity. With good find and replace tools, it is usually not difficult. Strong typing and testing will highlight anything you miss. Remember any fool can write code that a computer can understand, good programmers write code that humans can understand.

Lecture No. 43

Moving the amount calculation

As I look at amountOf, I can see that it uses information from the rental, but does not use information from the customer. This method is thus on the wrong object, it should be moved to the rental. To *move a method* you first copy the code over to rental, adjust it to fit in its new home and compile.

```
Class Rental
   double charge() {
      double result = 0;
      switch (tape().movie().priceCode()) {
         case Movie.REGULAR:
            result += 2;
            if (daysRented() > 2)
               result += (daysRented() - 2) * 1.5;
            break;
         case Movie.NEW_RELEASE:
            result += daysRented() * 3;
            break;
         case Movie.CHILDRENS:
            result += 1.5;
            if (daysRented() > 3)
               result += (daysRented() - 3) * 1.5;
            break;

      }
      return result;
   }
```

In this case fitting into its new home means removing the parameter.

The next step is to find every reference to the old method, and adjusting the reference to use the new method. In this case this step is easy as we just created the method and it is in only one place. In general, however, you need to do a find across all the classes that might be using that method.

```
public String statement() {
      double totalAmount = 0;
      int frequentRenterPoints = 0;
      Enumeration rentals = _rentals.elements();
      String result = "Rental Record for " + name() + "\n";
      while (rentals.hasMoreElements()) {
         double thisAmount = 0;
         Rental each = (Rental) rentals.nextElement();

         //determine amounts for each line
         thisAmount = each.charge();
```

```
        totalAmount += thisAmount;

        // add frequent renter points
        frequentRenterPoints ++;
        // add bonus for a two day new release rental
        if ((each.tape().movie().priceCode() == Movie.NEW_RELEASE) &&
each.daysRented() > 1) frequentRenterPoints ++;

        //show figures for this rental
        result += "\t" + each.tape().movie().name()+ "\t" + String.valueOf(thisAmount) +
"\n";

    }
    //add footer lines
    result +=  "Amount owed is " + String.valueOf(totalAmount) + "\n";
    result += "You earned " + String.valueOf(frequentRenterPoints) + " frequent renter
points";
    return result;

  }
```

When I've made the change the next thing is to remove the old method. The compiler should then tell me if I missed anything.

There is certainly some more I would like to do to Rental.charge() but I will leave it for the moment and return to Customer.statement().

The next thing that strikes me is that thisAmount() is now pretty redundant. It is set to the result of each.charge() and not changed afterwards. Thus I can eliminate thisAmount by *replacing a temp with a query*.

```
public String statement() {
    double totalAmount = 0;
    int frequentRenterPoints = 0;
    Enumeration rentals = _rentals.elements();
    String result = "Rental Record for " + name() + "\n";
    while (rentals.hasMoreElements()) {
        Rental each = (Rental) rentals.nextElement();

        //determine amounts for each line
        totalAmount += each.charge();

        // add frequent renter points
        frequentRenterPoints ++;
        // add bonus for a two day new release rental
        if ((each.tape().movie().priceCode() == Movie.NEW_RELEASE) &&
each.daysRented() > 1) frequentRenterPoints ++;

        //show figures for this rental
```

```
            result += "\t" + each.tape().movie().name()+ "\t" + String.valueOf(each.charge())
+ "\n";

        }
        //add footer lines
        result +=  "Amount owed is " + String.valueOf(totalAmount) + "\n";
        result += "You earned " + String.valueOf(frequentRenterPoints) + " frequent renter
points";
        return result;

    }
```

I like to get rid of temporary variables like thus as much as possible. Temps are often a problem in that they cause a lot of parameters to get passed around when they don't need to. You can easily lose track of what they are there for. They are particularly insidious in long methods. Of course there is a small performance price to pay, here the charge is now calculated twice. But it is easy to optimize that in the rental class, and you can optimize much more effectively when the code is properly refactored.

Extracting Frequent Renter Points

The next step is to do a similar thing for the frequent renter points. Again the rules vary with the tape, although there is less variation than with the charging. But it seems reasonable to put the responsibility on the rental. First we need to extract a method from the frequent renter points part of the code (highlighted below).

```
public String statement() {
        double totalAmount = 0;
        int frequentRenterPoints = 0;
        Enumeration rentals = _rentals.elements();
        String result = "Rental Record for " + name() + "\n";
        while (rentals.hasMoreElements()) {
            Rental each = (Rental) rentals.nextElement();

            //determine amounts for each line
            totalAmount += each.charge();

            // add frequent renter points
            frequentRenterPoints ++;
            // add bonus for a two day new release rental
            if ((each.tape().movie().priceCode() == Movie.NEW_RELEASE) &&
each.daysRented() > 1) frequentRenterPoints ++;

            //show figures for this rental
            result += "\t" + each.tape().movie().name()+ "\t" + String.valueOf(each.charge())
+ "\n";

        }
        //add footer lines
        result +=  "Amount owed is " + String.valueOf(totalAmount) + "\n";
```

```
        result += "You earned " + String.valueOf(frequentRenterPoints) + " frequent renter
points";
        return result;

    }
```

Again we look at the use of locally scoped variables. Again it uses each, which can be passed in as a parameter. The other temp used is frequentRenterPoints. In this case frequentRenterPoints does have a value beforehand. The body of the extracted method doesn't read the value, however, so we don't need to pass it in as a parameter as long as we use an appending assignment.

```
public String statement() {
        double totalAmount = 0;
        int frequentRenterPoints = 0;
        Enumeration rentals = _rentals.elements();
        String result = "Rental Record for " + name() + "\n";
        while (rentals.hasMoreElements()) {
            Rental each = (Rental) rentals.nextElement();

            //determine amounts for each line
            totalAmount += each.charge();

            // add frequent renter points
            frequentRenterPoints += frequentRenterPointOf(each);

            //show figures for this rental
            result += "\t" + each.tape().movie().name()+ "\t" + String.valueOf(each.charge())
+ "\n";

        }
        //add footer lines
        result +=  "Amount owed is " + String.valueOf(totalAmount) + "\n";
        result += "You earned " + String.valueOf(frequentRenterPoints) + " frequent renter
points";
        return result;

    }

int frequentRenterPointOf(Rental each) {
        if ((each.tape().movie().priceCode() == Movie.NEW_RELEASE) &&
each.daysRented() > 1) return 2;
        else return 1;
    }
```

I did the extraction, compiled and tested, and then did a move. With refactoring small steps are the best, that way less tends to go wrong.

```
public String statement() {
        double totalAmount = 0;
```

```
        int frequentRenterPoints = 0;
        Enumeration rentals = _rentals.elements();
        String result = "Rental Record for " + name() + "\n";
        while (rentals.hasMoreElements()) {
            Rental each = (Rental) rentals.nextElement();

            //determine amounts for each line
            totalAmount += each.charge();

            // add frequent renter points
            frequentRenterPoints += each.frequentRenterPoints();

            //show figures for this rental
            result += "\t" + each.tape().movie().name()+ "\t" + String.valueOf(each.charge())
+ "\n";

        }
        //add footer lines
        result +=  "Amount owed is " + String.valueOf(totalAmount) + "\n";
        result += "You earned " + String.valueOf(frequentRenterPoints) + " frequent renter
points";
        return result;

    }

int frequentRenterPoints() {
        if ((tape().movie().priceCode() == Movie.NEW_RELEASE) && daysRented() > 1)
return 2;
        else return 1;
    }
```

Removing Temps

As I suggested before, temporary variables can be a problem. They are only useful within their own routine, and thus they encourage long complex routines. In this case we have two temporary variables, both of which are being used to get a total from the rentals attached to the customer. Both the ascii and html versions will require these totals. I like to *replace temps with queries*. Queries are accessible to any method in the class, and thus encourage a cleaner design without long complex methods.

I began by replacing totalAmount with a charge() method on customer.

```
    public String statement() {
        double totalAmount = 0;
        int frequentRenterPoints = 0;
        Enumeration rentals = _rentals.elements();
        String result = "Rental Record for " + name() + "\n";
        while (rentals.hasMoreElements()) {
            Rental each = (Rental) rentals.nextElement();
```

```
            // add frequent renter points
            frequentRenterPoints += each.frequentRenterPoints();

            //show figures for this rental
            result += "\t" + each.tape().movie().name()+ "\t" + String.valueOf(each.charge())
+ "\n";

        }
        //add footer lines
        result +=  "Amount owed is " + String.valueOf(charge()) + "\n";
        result += "You earned " + String.valueOf(frequentRenterPoints) + " frequent renter
points";
        return result;

    }

    private double charge(){
        double result = 0;
        Enumeration rentals = _rentals.elements();
        while (rentals.hasMoreElements()) {
            Rental each = (Rental) rentals.nextElement();
            result += each.charge();
        }
        return result;
    }
```

After compiling and testing that refactoring, I then did the same for frequentRenterPoints.

```
    public String statement() {
        Enumeration rentals = _rentals.elements();
        String result = "Rental Record for " + name() + "\n";
        while (rentals.hasMoreElements()) {
            Rental each = (Rental) rentals.nextElement();
            //show figures for each rental
            result += "\t" + each.tape().movie().name()+ "\t" +
                      String.valueOf(each.charge()) + "\n";
        }
        //add footer lines
        result +=  "Amount owed is " + String.valueOf(charge()) + "\n";
        result += "You earned " + String.valueOf(frequentRenterPoints()) +
                  " frequent renter points";
        return result;
    }

    private int frequentRenterPoints() {
        int result = 0;
        Enumeration rentals = _rentals.elements();
        while (rentals.hasMoreElements()) {
            Rental each = (Rental) rentals.nextElement();
            result += each.frequentRenterPoints();
```

```
    }
    return result;
}
```

It is worth stopping and thinking a bit about this refactoring. Most refactoring reduce the amount of code, but this one increases it. That's because Java requires a lot of statements to set up a summing loop. Even a simple summing loop with one line of code per element needs six lines of support around it. Itís an idiom that is obvious to any programmer but it is noise that hides what the intent of the loop is. As Java develops and builds up its ability to handle block closures in the style of Smalltalk, I expect that overhead to decrease, probably to the single line that such an expression would take in Smalltalk.

The other concern with this refactoring lies in performance. The old code executed the while loop once, the new code executes it three times. If the while loop takes time, this might significantly impair performance. Many programmers would not do this refactoring simply for this reason. But note the words "if" and "might". While some loops do cause performance issues, most do not. So while refactoring donít worry about this. When you optimize you will have to worry about it, but you will then be in a much better position to do something about it, and you will have more options to optimize effectively. (For a good discussion on why it is better to write clearly first and then optimize, see [McConnell, Code Complete].

These queries are now available to any code written in the customer class. Indeed they can easily be added to the interface of the class should other parts of the system need this information. Without queries like these, other methods need to deal with knowing about the rentals and building the loops. In a complex system that will lead to much more code to write and maintain.

You can see the difference immediately with the htmlStatement(). I am now at the point where I take off my refactoring hat and put on my adding function hat. I can write htmlStatement()like this (and add an appropriate test).

```
public String htmlStatement() {
    Enumeration rentals = _rentals.elements();
    String result = "<H1>Rentals for <EM>" + name() + "</EM></H1><P>\n";
    while (rentals.hasMoreElements()) {
        Rental each = (Rental) rentals.nextElement();
        //show figures for each rental
        result += each.tape().movie().name()+ ": " +
                String.valueOf(each.charge()) + "<BR>\n";
    }
    //add footer lines
    result +=  "<P>You owe <EM>" + String.valueOf(charge()) + "</EM><P>\n";
    result += "On this rental you earned <EM>" +
String.valueOf(frequentRenterPoints()) + "</EM> frequent renter points<P>";
    return result;

}
```

There is still some code copied from the ascii version, but that is mainly due to setting up the loop. Further refactoring could clean that up further, extracting methods for header, footer, and detail line are one route I could take. But that isnít where I want to spend my time, I would like to move onto the methods Iíve moved onto rental. Back on with the refactoring hat.

Moving the Rental Calculations to Movie

Yes itís that switch statement that is bugging me. It is a bad idea to do a switch based on an attribute of another object. If you must use a switch statement, it should be on your own data, not on someone elseís. This implies that the charge should move onto movie

```
Class movie Ö
   double charge(int daysRented) {
      double result = 0;
      switch (priceCode()) {
         case REGULAR:
            result += 2;
            if (daysRented > 2)
               result += (daysRented - 2) * 1.5;
            break;
         case NEW_RELEASE:
            result += daysRented * 3;
            break;
         case CHILDRENS:
            result += 1.5;
            if (daysRented > 3)
               result += (daysRented - 3) * 1.5;
            break;

      }
      return result;
   }
```

For this to work I have to pass in the length of the rental, which of course is data due of the rental. The method effectively uses two pieces of data, the length of the rental and the type of the movie. Why do I prefer to pass the length of rental rather than the movieís type? Its because type information tends to be more volatile. I can easily imagine new types of videos appearing. If I change the movieís type I want the least ripple effect, so I prefer to calculate the charge within the movie.

I compiled the method into movie and then adjusted the charge method on rental to use the new method.

```
Class rentalÖ
   double charge() {
      return _tape.movie().charge(_daysRented);
   }
```

Some people would prefer to remove that chain of calls by having a charge(int) message on tape. This would lead to

```
Class rental
   double charge() {
      return _tape.charge(_daysRented);
   }

Class tape
   double charge() {
      return _movie.charge(_daysRented);
   }
```

You can make that change if you like, I donít tend to worry about message chains providing that they all lie in the same package. If they cross package boundaries, then Iím not so happy, and would add an insulating method.

Having done this with charge amounts, Iím inclined to do the same with frequent renter points. The need is less pressing, but I think it is more consistent to do them both the same way. And again if the movie classifications change it makes it easier to update the code.

```
Class rentalÖ
   int frequentRenterPoints() {
      return _tape.movie().frequentRenterPoints(_daysRented);
   }

class movieÖ
   int frequentRenterPoints(int daysRented){
      if ((priceCode() == NEW_RELEASE) && daysRented > 1) return 2;
      else return 1;
   }
```

With these two changes I can hide those constants, which is generally a Good Thing. Even constant data should be private.

```
   private static final int  CHILDRENS = 2;
   private static final int  REGULAR = 0;
   private static final int  NEW_RELEASE = 1;
```

To really do this, however, I need to change a couple of other parts of the class. I need to change how we create a movie. I used to create a movie with a message like

```
new Movie ("Ran", Movie.REGULAR);
```

and the constructor

```
class MovieÖ
            private Movie(String name, int priceCode) {
                        _name = name;
```

```
                _priceCode = priceCode;
        }
```

To keep this type code hidden I need some creation methods.

```
public static Movie newNewRelease(String name){
   return new Movie (name, NEW_RELEASE);
}
public static Movie newRegular(String name){
   return new Movie (name, REGULAR);
}
public static Movie newChildrens(String name) {
   return new Movie (name, CHILDRENS);
}
```

Now I create a new movie with

```
Movie.newRegular("Monty Python and the Holy Grail");
```

Movies can change their classification. I change a movieís classification with

```
aMovie.setPriceCode(Movie.REGULAR);
```

I will need to add a bunch of methods to handle the changes of classification.

```
public void beRegular() {
   _priceCode = REGULAR;
}

public void beNewRelease() {
   _priceCode = NEW_RELEASE;
}

public void beChildrens() {
   _priceCode = CHILDRENS;
}
```

It is a bit of effort to set up these methods, but they are a much more explicit interface then the type codes. Just looking at the name of the method tells you what kind of movie you are getting. This makes the code more understandable. The trade off is that each time I add a price code I have to add a creation and update method. If I had lots of price codes this would hurt (so I wouldnít do it). If I have a few, however, then it is quite reasonable.

At lastÖ inheritance

So we have several types of movie, which have different ways of answering the same question. This sounds like a job for subclasses. We could have three subclasses of movie, each of which can have its own version of charge.

This would allow me to replace the switch statement by using polymorphism. Sadly it has one slight flaw: it doesn't work. A move can change its classification during its lifetime. An object cannot change its class during its lifetime. There is a solution however, the *state pattern* [Gang of Four]. With the state pattern the classes look like this.

By adding the indirection we can do the subclassing from the price code object, changing the price whenever we need to.

With a complex class you have to move data and methods around in small pieces to avoid errors, it seems slow but it is the quickest because you avoid debugging. For this case I could probably move the data and methods in one go as the whole thing is not too complicated. However Iíll do it the bit by bit way, so you can see how it goes. Just remember to do it one small bit at a time if you do this to a complicated class.

The first step is to create the new classes. Then I need to sort out how they are managed. As the diagram shows they are all singletons. It seems sensible to get hold of them via the superclass with a method like Price.regular(). I can do this by getting the superclass to manage the instances of the subclasses.

```
abstract class Price {
   static Price regular() {
      return _regular;
   }

   static Price childrens() {
      return _childrens;
   }
   static Price newRelease() {
      return _newRelease;
   }

   private static Price _childrens = new ChildrensPrice();
   private static Price _newRelease = new NewReleasePrice();
   private static Price _regular = new RegularPrice();
}
```

Now I can begin to move the data over. The first piece of data to move over is the price code. Of course Iím not actually going to use the price code within the Price object, but I will give it the illusion of doing so. That way the old methods will still work. They key is to modify those methods that access and update the price code value within Movie. My first step is to *self-encapsulate the type code*, ensuring that all uses of the type code go though getting and setting methods. Since most of the code came from other classes, most methods already use the getting method. However the constructors do access the price code, I can use the setting methods instead.

```
        public static Movie newNewRelease(String name){
    Movie result = new Movie (name);
    result.beNewRelease();
    return result;
        }
        public static Movie newRegular(String name){
    Movie result = new Movie (name);
    result.beRegular();
    return result;
        }
        public static Movie newChildrens(String name) {
    Movie result = new Movie (name);
    result.beChildrens();
    return result;
        }

        private Movie(String name) {
                _name = name;
        }
```

After compiling and testing I now change getting and setting methods to use the new class.

```
  public void beRegular() {
     _price = Price.regular();
  }

  public void beNewRelease() {
     _price = Price.newRelease();
  }

  public void beChildrens() {
    _price = Price.childrens();
  }
  public int priceCode() {
     return _price.priceCode();
  }
```

And provide the priceCode methods on Price and its subclasses.

```
Class PriceÖ
  abstract int priceCode();

Class RegularPriceÖ
   int priceCode(){
     return Movie.REGULAR;
   }
```

To do this I need to make the constants non-private again. This is fine, I donít mind them having a little fame before they bite the dust.

I can now compile and test and the more complex methods donít realize the world has changed.

After moving the data I can now start moving methods. My prime target is the charge() method. It is simple to move.

```
Class MovieÖ
   double charge(int daysRented) {
      return _price.charge(daysRented);
   }

Class PriceÖ
   double charge(int daysRented) {
      double result = 0;
      switch (priceCode()) {
         case Movie.REGULAR:
            result += 2;
            if (daysRented > 2)
               result += (daysRented - 2) * 1.5;
            break;
         case Movie.NEW_RELEASE:
            result += daysRented * 3;
            break;
         case Movie.CHILDRENS:
            result += 1.5;
            if (daysRented > 3)
               result += (daysRented - 3) * 1.5;
            break;

      }
      return result;
   }
```

Once it is moved I can start *replacing the case statement with inheritance*. I do this by taking one leg of the case statement at a time, and creating an overriding method. I start with RegularPrice.

```
Class RegularPriceÖ
   double charge(int daysRented){
      double result = 2;
      if (daysRented > 2)
         result += (daysRented - 2) * 1.5;
      return result;
   }
```

This will override the parent case statement, which I just leave as it is. I compile and test for this case, then take the next leg, compile and testÖ. (To make sure Iím executing the subclass code, I like to throw in a deliberate bug and run it to ensure the tests blow up. Not that Iím paranoid or anything.)

```
Class ChildrensPrice
   double charge(int daysRented){
      double result = 1.5;
      if (daysRented > 3)
         result += (daysRented - 3) * 1.5;
      return result;
   }

Class NewReleasePriceÖ
   double charge(int daysRented){
      return daysRented * 3;
   }
```

When Iíve done that with all the legs, I declare the Price.charge() method abstract.

```
Class PriceÖ
   abstract double charge(int daysRented);
```

I can now do the same procedure with frequentRenterPoints(). First I move the method over to Price.

```
Class MovieÖ
   int frequentRenterPoints(int daysRented){
      return _price.frequentRenterPoints(daysRented);
   }
Class PriceÖ
   int frequentRenterPoints(int daysRented){
      if ((priceCode() == Movie.NEW_RELEASE) && daysRented > 1) return 2;
      else return 1;
   }
```

In this case, however I wonít make the superclass method abstract. Instead I will create an overriding method for new releases, and leave a defined method (as the default) on the superclass.

```
Class NewReleasePrice
   int frequentRenterPoints(int daysRented){
      return (daysRented > 1) ?
         2:
         1;
   }

Class PriceÖ
   int frequentRenterPoints(int daysRented){
      return 1;
   }
```

Now I have removed all the methods that needed a price code. So I can get rid of the price code methods and data on both Movie and Price.

Putting in the state pattern was quite an effort, was it worth it? The gain is now that should I change any of prices behavior, add new prices, or add extra price dependent behavior; it will be much easier to change. The rest of the application does not know about the use of the state pattern. For the tiny amount of behavior I currently have it is not a big deal. But in a more complex system with a dozen or so price dependent methods this would make a big difference. All these changes were small steps, it seems slow to write it like this, but not once did I have to open the debugger. So the process actually flowed quite quickly.

© Virtual University of Pakistan

Lecture No. 44

Capability Maturity Model Integration (CMMI)

Capability Maturity Model or CMM is a reference model of mature practices in a specified discipline, used to assess a group's capability to perform that discipline. In fact there are a number of CMMs. They differ by discipline (software, systems, acquisition, etc.), structure (staged versus continuous), how maturity is defined (process improvement path), and how capability is defined (institutionalization). Hence "Capability Maturity Model®" and CMM® are used by the Software Engineering Institute (SEI) to denote a particular class of maturity models.

Software CMM, the one we saw earlier on, is one of the many CMM models.

Name	Structure	Domain
Software CMM	staged	software development
System Engineering CMM	continuous	system engineering
Software Acquisition CMM	staged	software acquisition
System Security Engineering CMM	continuous	security engineering
FAA-iCMM	continuous	software engineering, systems engineering, and acquisition
IPD-CMM	hybrid	integrated product development
People CMM	staged	workforce
SPICE Model	continuous	software development

Since these models have different structure and application domains, an organization could potentially use many of these models for their different activities, at times it could become problematic for them.

CMMI Integrates systems and software disciplines into single process improvement framework and provides a framework for introducing new disciplines as needs arise. It can now be applied to just the software engineering projects in an organization or for the entire spectrum of activities outlined above.

CMMI Representations

A representation allows an organization to pursue different improvement objectives. There are two types of representations in the CMMI models: staged and continuous. The organization and presentation of the data are different in each representation. However, the content is the same.

Staged Representation

Staged representation is classical representation we have already seen previously. It:

- Provides a proven sequence of improvements, each serving as a foundation for the next.
- Permits comparisons across and among organizations by the use of maturity levels.

- Provides an easy migration from the SW-CMM to CMMI.
- Provides a single rating that summarizes appraisal results and allows comparisons among organizations

This representation indicates maturity of an organization's standard process -- to answer, "What is a good order for approaching improvement across the organization?"

You may recall that a maturity level is a well-defined evolutionary plateau of process improvement. There are five maturity levels and each level is a layer in the foundation for continuous process improvement using a proven sequence of improvements, beginning with basic management practices and progressing through a predefined and proven path of successive levels.

Each maturity level provides a necessary foundation for effective implementation of processes at the next level. Higher level processes have less chance of success without the discipline provided by lower levels. The effect of innovation can be obscured in a noisy process.

Higher maturity level processes may be performed by organizations at lower maturity levels, with the risk of not being consistently applied in a crisis.

Continuous Representation

Continuous representation allows you to select the order of improvement that best meets your organization's business objectives and mitigates your organization's areas of risk. It enables comparisons across and among organizations on a process-area-by-process-area basis and provides an easy migration from EIA 731 (and other models with a continuous representation) to CMMI.

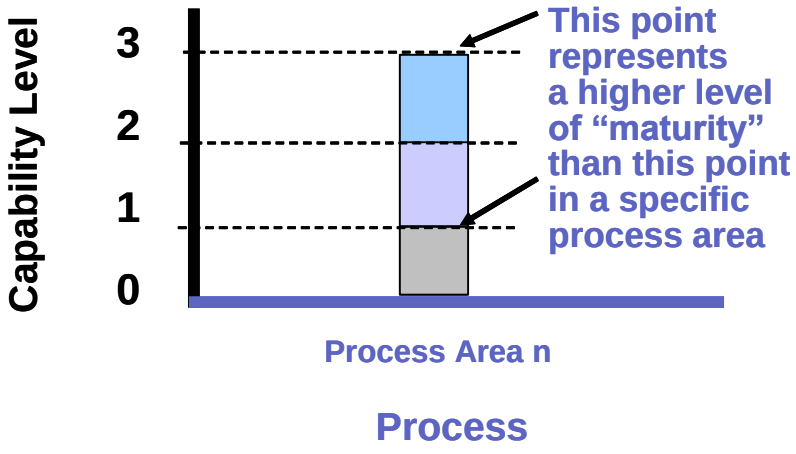
As opposed to the staged representation, it indicates improvement within a single process area -- to answer, "What is a good order for approaching improvement of this process area?"

Capability Levels

A capability level is a well-defined evolutionary plateau describing the organization's capability relative to a process area. There are six capability levels. For capability levels 1-5, there is an associated generic goal. Each level is a layer in the foundation for continuous process improvement. Thus, capability levels are cumulative, i.e., a higher capability level includes the attributes of the lower levels. The five (actually six) capability levels (starting from 0) are enumerated below in the reverse order, 5 being the highest and 0 being the lowest.

5 Optimizing
4 Quantitatively Managed
3 Defined
2 Managed
1 Performed
0 Incomplete

The process area capability of an implemented process can be represented by a bar as shown below.



Relating Process Area Capability and Organizational Maturity

Organizational maturity is the focus of the staged representation, whereas process area capability is the focus of the continuous representation.

Organizational maturity and process area capability are similar concepts. The difference between them is that organizational maturity pertains to a set of process areas across an organization, while process area capability deals with a set of processes relating to a single process area or specific practice.

Comparison of Representations

Staged

- Process improvement is measured using maturity levels.
- Maturity level is the degree of process improvement across a predefined set of process areas.
- Organizational maturity pertains to the "maturity" of a set of processes across an organization

Continuous

- Process improvement is measured using capability levels.
- Capability level is the achievement of process improvement within an individual process area.
- Process area capability pertains to the "maturity" of a particular process across an organization.

Advantages of Each Representation

Staged provides a roadmap for implementing groups of process areas and sequencing of implementation. It has a familiar structure for those transitioning from the Software CMM.

Continuous provides maximum flexibility for focusing on specific process areas according to business goals and objectives and has a familiar structure for those transitioning from EIA 731.

As the staged representation requires all KPAs to be addressed at a particular level before a company can move to the next maturity level, it may not be easy for small companies to implement this model. There may be a number of activities that may not be relevant to their type of work but they would still have to do them in order to be at a certain level. On the other hand, organization can focus on their own areas of expertise and may be able to achieve high capability levels in some areas without bothering about the rest. This is a great advantage for small organization and hence this model is believed to be more suitable for small Pakistani organizations than the staged one.

Lecture No. 45

CMM Overview

CMM Maturity Levels

There are five levels defined along the continuum of the CMM and, according to the SEI: "Predictability, effectiveness, and control of an organization's software processes are believed to improve as the organization moves up these five levels. While not rigorous, the empirical evidence to date supports this belief."

Level 1 - Ad hoc (Chaotic)

It is characteristic of processes at this level that they are (typically) undocumented and in a state of dynamic change, tending to be driven in an *ad hoc*, uncontrolled and reactive manner by users or events. This provides a chaotic or unstable environment for the processes.

Organizational implications

(a) Institutional knowledge tends to be scattered (there being limited structured approach to knowledge management) in such environments, not all of the stakeholders or participants in the processes may know or understand all of the components that make up the processes. As a result, process performance in such organizations is likely to be variable (inconsistent) and depend heavily on the institutional knowledge, or the competence, or the heroic efforts of relatively few people or small groups.

(b) Despite the chaos, such organizations manage to produce products and services. However, in doing so, there is significant risk that they will tend to exceed any estimated budgets or schedules for their projects - it being difficult to estimate what a process will do when you do not fully understand the process (what it is that you do) in the first place and cannot therefore control it or manage it effectively.

(c) Due to the lack of structure and formality, organizations at this level may over-commit, or abandon processes during a crisis, and it is unlikely that they will be able to repeat past successes. There tends to be limited planning, limited executive commitment or buy-in to projects, and limited acceptance of processes.

Level 2 - Repeatable

It is characteristic of processes at this level that some processes are repeatable, possibly with consistent results. Process discipline is unlikely to be rigorous, but where it exists it may help to ensure that existing processes are maintained during times of stress.

Organizational implications

i) Processes and their outputs could be visible to management at defined points, but results may not always be consistent. For example, for project/program management processes, even though (say) some basic processes are established to track cost, schedule, and functionality, and if a degree of process discipline is in place to repeat earlier successes on projects with similar applications and scope, there could still be a significant risk of exceeding cost and time estimates.

Level 3 - Defined

It is characteristic of processes at this level that there are sets of defined and documented standard processes established and subject to some degree of improvement over time. These standard processes are in place (i.e., they are the AS-IS processes) and used to establish consistency of process performance across the organization.

Organizational implications

(a) Process management starts to occur using defined documented processes, with mandatory process objectives, and ensures that these objectives are appropriately addressed.

Level 4 - Managed

It is characteristic of processes at this level that, using process metrics, management can effectively control the AS-IS process (e.g., for software development). In particular, management can identify ways to adjust and adapt the process to particular projects without measurable losses of quality or deviations from specifications. Process Capability is established from this level.

Organizational implications

a) Quantitative quality goals tend to be set for process output - e.g., software or software maintenance.
b) Using quantitative/statistical techniques, process performance is measured and monitored and generally predictable and controllable also.

Level 5 - Optimizing

It is a characteristic of processes at this level that the focus is on continually improving process performance through both incremental and innovative technological changes/improvements.

Organizational implications

(a) Quantitative process-improvement objectives for the organization are established, continually revised to reflect changing business objectives, and used as criteria in managing process improvement. Thus, process improvements to address common causes of process variation and measurably improve the organization’s processes are identified, evaluated, and deployed.

(b) The effects of deployed process improvements are measured and evaluated against the quantitative process-improvement objectives.
(c) Both the defined processes and the organization's set of standard processes are targets for measurable improvement activities.
(d) A critical distinction between maturity level 4 and maturity level 5 is the type of process variation addressed.

At maturity level 4, processes are concerned with addressing statistical *special causes* of process variation and providing statistical predictability of the results, and though processes may produce predictable results, the results may be insufficient to achieve the established objectives.
At maturity level 5, processes are concerned with addressing statistical *common causes* of process variation and changing the process (for example, shifting the mean of the process performance) to improve process performance. This would be done at the same time as maintaining the likelihood of achieving the established quantitative process-improvement objectives.

Extensions

Some versions of CMMI from SEI indicate a "level 0", characterized as "Incomplete". Some pundits leave this level out as redundant or unimportant, but Pressman and others make note of it.

Comparison of CMMI Representations Staged Continuous

What is a CMMI model representation? The answer requires an explanation of the structure of CMMI models. The basic building blocks in every CMMI model are called "process areas." A process area does not describe *how* an effective process is executed (e.g., entrance and exit criteria, roles of participants, resources). Instead, a process area describes *what* those using an effective process do (practices) and *why* they do those things (goals).

Selecting a Representation

When making the decision about which architectural representation to use for process improvement, comparative advantages of each approach as represented in the given Table.

Continuous Representation	**Staged Representation**
Grants explicit freedom to select the order of improvement that best meets the organization's business objectives. Enables increased visibility into the capability achieved within each individual process area. Supports a focus on risks specific to	Enables organizations to have a predefined and proven path. Builds on a relatively long history of use. Case studies and data exist that show return on investment. Permits comparisons across and among

individual process areas. Affords a more direct comparison of process improvement to ISO 15504 because the organization of process areas is derived from 15504. Allows the generic practices from higher capability levels to be more evenly and completely applied to all of the process areas.	organizations. Introduces a sequence of improvements, beginning with basic management practices and progressing through successive levels, each serving as a foundation for the next. Summarizes process-improvement results in a simple form—a single maturity-level number.

Project Management Concerns

Project management is the discipline of planning, organizing and managing resources to bring about the successful completion of specific project goals and objectives.

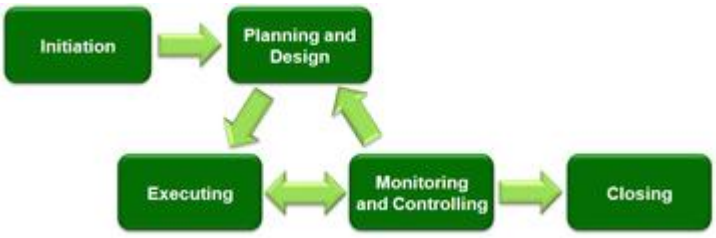
A project is a finite endeavor (having specific start and completion dates) undertaken to create a unique product or service which brings about beneficial change or added value. This finite characteristic of projects stands in sharp contrast to processes, or operations, which are permanent or semi-permanent functional work to repetitively produce the same product or service. In practice, the management of these two systems is often found to be quite different, and as such requires the development of distinct technical skills and the adoption of separate management.

There are several approaches that can be taken to managing project activities including agile, interactive, incremental, and phased approaches.

Regardless of the approach employed, careful consideration needs to be given to clarify surrounding project objectives, goals, and importantly, the roles and responsibilities of all participants and stakeholders.

The traditional approach

A traditional phased approach identifies a sequence of steps to be completed. In the "traditional approach", we can distinguish 5 components of a project (4 stages plus control) in the development of a project:



Typical development phases of a project

- Project initiation stage;
- Project planning or design stage;
- Project execution or production stage;
- Project monitoring and controlling systems;
- Project completion stage.

Not all the projects will visit every stage as projects can be terminated before they reach completion. Some projects probably don't have the planning and/or the monitoring. Some projects will go through steps 2, 3 and 4 multiple times.

Many industries utilize variations on these stages. For example, in bricks and mortar architectural design, projects typically progress through stages like Pre-Planning, Conceptual Design, Schematic Design, Design Development, Construction Drawings (or Contract Documents), and Construction Administration. In software development, this approach is often known as "waterfall development", i.e., one series of tasks after another in linear sequence. In software development many organizations have adapted the Rational Unified Process (RUP) to fit this methodology, although RUP does not require or explicitly recommend this practice. Waterfall development can work for small tightly defined projects, but for larger projects of undefined or unknowable scope, it is less suited. The Cone of Uncertainty explains some of this as the planning made on the initial phase of the project suffers from a high degree of uncertainty. This becomes specially true as software development is often the realization of a new or novel product, this method has been widely accepted as ineffective for software projects where requirements are largely unknowable up front and susceptible to change. While the names may differ from industry to industry, the actual stages typically follow common steps to problem solving — "defining the problem, weighing options, choosing a path, implementation and evaluation."

Software Quality Assurance

Software quality assurance (SQA) consists of a means of monitoring the software engineering processes and methods used to ensure quality. The methods by which this is accomplished are many and varied, and may include ensuring conformance to one or more standards, such as ISO 9000 or CMMI.

This definition emphasizes upon three important points:

- Software requirements are the foundation from which quality is measured. Lack of conformance is lack of quality
- Specified standards define a set of development criteria that guide the manner in which software is engineered. If the criteria are not followed, lack of quality will almost surely result.
- A set of implicit requirements often goes unmentioned (ease of use, good maintainability etc.)

Another very important question is: Do you need to worry about it after the code has been generated? In fact, SQA is an umbrella activity that is applied throughout the software process.

Quality Assurance

Goal of quality assurance is to provide the management with the necessary data to be informed about product quality. It consists of auditing and reporting functions of management. If data provided through QA identifies problems, the management deploys the necessary resources to fix it and hence achieves desired quality control.

Cost of quality

A very significant question is: does quality assurance add any value. That is, is worth spending a lot of money in quality assurance practices? In order to understand the impact of quality assurance practices, we have to understand the cost of quality (or lack thereof) in a system.
Quality has a direct and indirect cost in the form of cost of prevention, appraisal, and failure.
If we try to prevent problems, obviously we will have to incur cost. This cost includes:

- Quality planning
- Formal technical reviews
- Test equipment
- Training

We will discuss these in more detail in the later sections.
The cost of appraisal includes activities to gain insight into the product condition. It involves in-process and inter-process inspection and testing.
And finally, failure cost. Failure cost has two components: internal failure cost and external failure cost. Internal failure cost requires rework, repair, and failure mode analysis. On the other hand, external failure cost involves cost for complaint resolution, product return and replacement, help-line support, warranty work, and law suits.

SQA Activities

There are two different groups involved in SQA related activities:

- Software engineers who do the technical work
- SQA group who is responsible for QA planning, oversight, record keeping, analysis, and reporting

Software engineers address quality by applying solid technical methods and measures, conducting formal and technical reviews, and performing well planned software testing. The SQA group assists the software team in achieving a high quality product.

SQA Group Activities

An SQA plan is developed for the project during project planning and is reviewed by all stake holders. The plan includes the identification of:

- Evaluations to be performed
- Audits and reviewed to be performed
- Standards that are applicable to the project
- Procedures for error reporting and tracking

- Documents to be produced by the SQA group
- Amount of feedback provided to the software project team

The group participates in the development of the project's software process description. The software team selects the process and SQA group reviews the process description for compliance with the organizational policies, internal software standards, externally imposed standards, and other parts of the software project plan.

Quality Control

The next question that we need to ask is, once we have defined how to assess quality, how are we going to make sure that our processes deliver the product with the desired quality. That is, how are we going to control the quality of the product?
The basic principle of quality control is to control the variation as variation control is the heart of quality control. It includes resource and time estimation, test coverage, variation in number of bugs, and variation in support.
From one project to another we want to minimize the predicted resources needed to complete a project and calendar time. This involves a series of inspection, reviews, and tests and includes feedback loop. So quality control is a combination of measurement and feedback and combination of automated tools and manual interaction.

The End