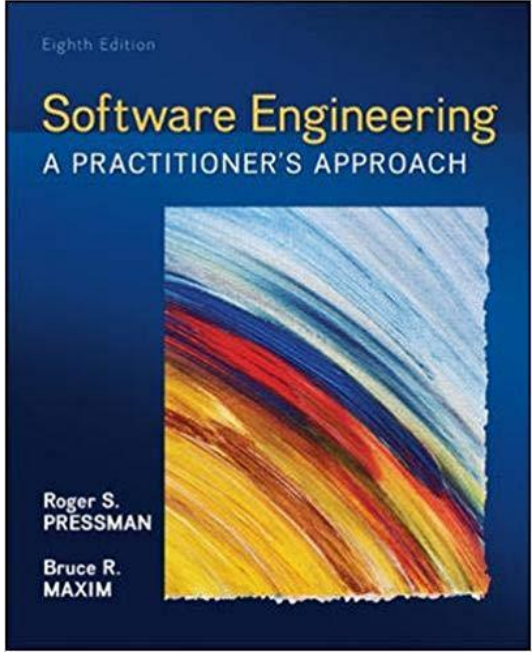


Software Quality

CS46 Software Engineering

Dr. Shilpa chaudhari

Department of Computer Science and Engineering
RIT, Bangalore

Outline of Unit-4

- **Quality Concepts:** Software Quality, Software Quality Dilemma, Achieving Software Quality - 19.2, 19.3, 19.4
- Formal Technical Reviews -20.6
- **Software Project Estimation** - Observations on Estimation, The Project Planning Process Software Project Estimation, Decomposition Techniques, Empirical Estimation Models, Estimation for Object-Oriented Projects 33.1, 33.2, 33.5, 33.6, 33.7, 33.8
- Project Scheduling –Scheduling. 34.5
- **Risk Management:** Reactive versus Proactive Risk Strategies, Software Risks, Risk Identification, Risk Projection, Risk Refinement, Risk Mitigation, Monitoring, and Management. 35.1, 35.2, 35.3, 35.4, 35.5, 34.6
- Computation of relevant metrics for the case study on current problem statement of software development considered in Unit-1.
- **Software Maintenance**, Software Supportability, Software Reengineering, Reverse Engineering, Restructuring, Forward Engineering. 36.2,36.5, 36.6, 36.7, 36.8

Software Quality Addresses

- What are the generic characteristics of high-quality software?
- How do we review quality and how are effective reviews conducted?
- What is software quality assurance?
- What strategies are applicable for software testing?
- What methods are used to design effective test cases?
- Are there realistic methods that will ensure that software is correct?
- How can we manage and control changes that always occur as software is built?
- What measures and metrics can be used to assess the quality of requirements and design models, source code, and test cases?

Software Quality

- To achieve high-quality software, four activities must occur:
  - Proven software engineering process and practice
  - solid project management
  - Comprehensive quality control
  - the presence of a quality assurance infrastructure
- *Definition: An effective software process applied in a manner that creates a useful product that provides measurable value for those who produce it and those who use it*
- The definition serves to emphasize three important points
  - *effective software process*
  - *useful product*
  - *adding value for both the producer and user*

Effective Software Process

- Establishes the infrastructure that supports any effort at building a high quality software product.
- The management aspects of process create the checks and balances that help avoid project chaos—a key contributor to poor quality.
- Software engineering practices allow the developer to analyze the problem and design a solid solution—both critical to building high quality software.
- Finally, umbrella activities such as change management and technical reviews have as much to do with quality as any other part of software engineering practice.

Useful Product

- delivers the content, functions, and features that the end-user desires
- But as important, it delivers these assets in a reliable, error free way.
- A useful product always satisfies those requirements that have been explicitly stated by stakeholders.
- In addition, it satisfies a set of implicit requirements (e.g., ease of use) that are expected of all high quality software.

Adding value for both the producer and user

- By *adding value for both the producer and user* of a software product, high quality software provides benefits for the software organization and the end-user community.
- The software organization gains added value because high quality software requires less maintenance effort, fewer bug fixes, and reduced customer support.
- The user community gains added value because the application provides a useful capability in a way that expedites some business process.
- The end result is:
  - (1) greater software product revenue,
  - (2) better profitability when an application supports a business process, and/or
  - (3) improved availability of information that is crucial for the business.

Quality views

- David Garvin factors
- McCall's Quality Factors
- ISO 9126 Quality Factors
- Targeted Factors

David Garvin Factors

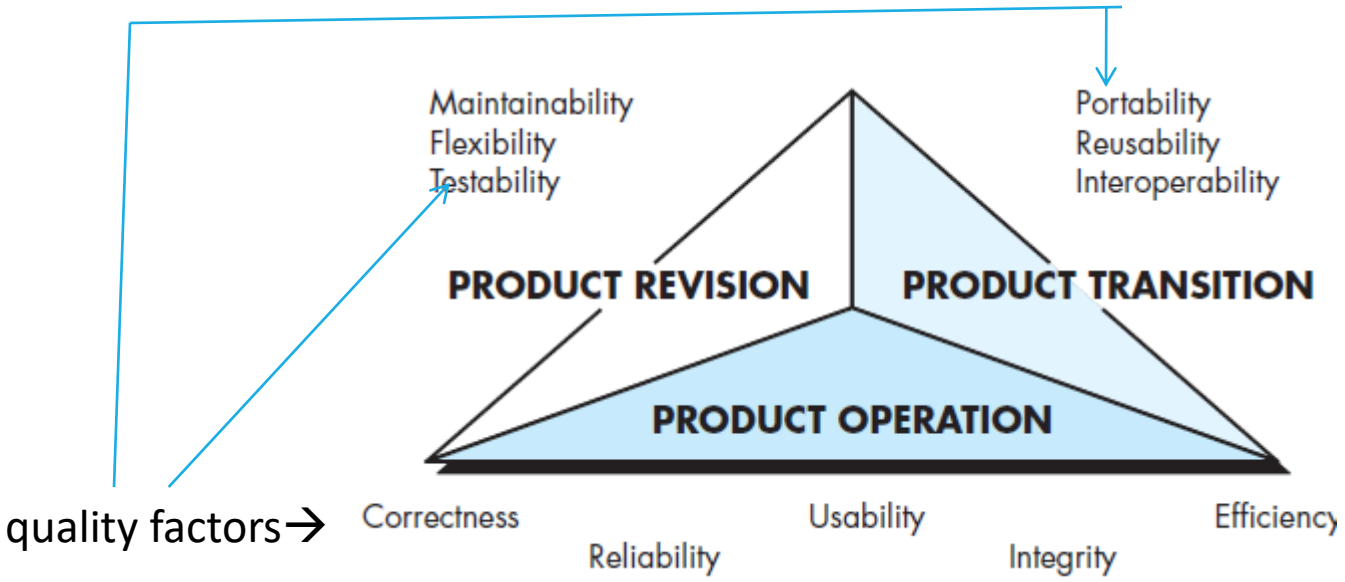
- David Garvin suggests that quality should be considered by taking a multidimensional viewpoint that
  - begins with an assessment of conformance
  - terminates with a transcendental (aesthetic) view
- Garvin's eight dimensions of quality were not developed specifically for software, they can be applied when software quality is considered:
  - **Performance Quality.**
  - **Feature quality.**
  - **Reliability.**
  - **Conformance.**
  - **Durability**
  - **Serviceability**
  - **Aesthetics**
  - **Perception**

David Garvin Factors

- David Garvin [Gar87]:
  - **Performance Quality.** Does the software deliver all content, functions, and features that are specified as part of the requirements model in a way that provides value to the end-user?
  - **Feature quality.** Does the software provide features that surprise and delight first-time end-users?
  - **Reliability.** Does the software deliver all features and capability without failure? Is it available when it is needed? Does it deliver functionality that is error free?
  - **Conformance.** Does the software conform to local and external software standards that are relevant to the application? Does it conform to de facto design and coding conventions? For example, does the user interface conform to accepted design rules for menu selection or data input?
  - **Durability.** Can the software be maintained (changed) or corrected (debugged) without the inadvertent generation of unintended side effects? Will changes cause the error rate or reliability to degrade with time?
  - **Serviceability.** Can the software be maintained (changed) or corrected (debugged) in an acceptably short time period. Can support staff acquire all information they need to make changes or correct defects?
  - **Aesthetics.** Most of us would agree that an aesthetic entity has a certain elegance, a unique flow, and an obvious "presence" that are hard to quantify but evident nonetheless.
  - **Perception.** In some situations, you have a set of prejudices that will influence your perception of quality.

McCall's Quality Factors

- focus on three important aspects of a software product:
  - its operational characteristics
  - its ability to undergo change
  - its adaptability to new environments



McCall's Quality Factors

- *Correctness. The extent to which a program satisfies its specification 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 and code 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 for, and interpret output of* a program.
- *Maintainability. Effort required to locate and fi x 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.
- *Portability. Effort required to transfer the program from one hardware and/or software* system environment to another.
- *Reusability. Extent to which a program [or parts of a program] can be reused in other* applications—related to the packaging and scope of the functions that the program performs.
- *Interoperability. Effort required to couple one system to another.*

ISO 9126 Quality Factors

- identifies six key quality attributes for computer software
  - **Functionality -**The degree to which the software satisfies stated needs
    - Sub-attributes: suitability, accuracy, interoperability, compliance, and security.
  - **Reliability -** The amount of time that the software is available for use
    - Sub-attributes: maturity, fault tolerance, recoverability.
  - **Usability -** ***The degree to which the software is easy to use***
    - Sub-attributes: understandability, learnability, operability.
  - **Efficiency. The degree to which the software makes optimal use of system** resources
    - Sub-attributes: time behavior, resource behavior.
  - **Maintainability. The ease with which repair may be made to the software**
    - Sub-attributes: analyzability, changeability, stability, testability.
  - **Portability. The ease with which the software can be transposed from one** environment to another
    - Sub-attributes: adaptability, installability, conformance, replaceability

Targeted Quality Factors

- Previous factors focus on the software as a whole and can be used as a generic indication of the quality of an application.
- A software team can develop a set of quality characteristics and associated questions that would probe the degree to which each factor has been satisfied.
- For example- *usability*
  - If you were asked to review a user interface and assess its usability, how would you proceed?
  - To conduct your assessment, you'll need to address specific, measurable (or at least, recognizable) attributes of the interface.
    - **Intuitiveness**
    - **Efficiency**
    - **Robustness**
    - **Richness**

Targeted Quality Factors

- **Intuitiveness. The degree to which the interface follows expected usage patterns** so that even a novice can use it without significant training.
  - Is the interface layout conducive to easy understanding?
  - Are interface operations easy to locate and initiate?
  - Does the interface use a recognizable metaphor?
  - Is input specified to economize key strokes or mouse clicks?
  - Does the interface follow the three golden rules?
  - Do aesthetics aid in understanding and usage?

Targeted Quality Factors

- **Efficiency. The degree to which operations and information can be located or** initiated.
  - Does the interface layout and style allow a user to locate operations and information efficiently?
  - Can a sequence of operations (or data input) be performed with an economy of motion?
  - Are output data or content presented so that it is understood immediately?
  - Have hierarchical operations been organized in a way that minimizes the depth to which a user must navigate to get something done?

Targeted Quality Factors

- **Robustness. The degree to which the software handles bad input data or inappropriate** user interaction.
  - Will the software recognize the error if data values are at or just outside prescribed input boundaries? More importantly, will the software continue to operate without failure or degradation?
  - Will the interface recognize common cognitive or manipulative mistakes and explicitly guide the user back on the right track?
  - Does the interface provide useful diagnosis and guidance when an error condition (associated with software functionality) is uncovered?

Targeted Quality Factors

- **Richness. The degree to which the interface provides a rich feature set.**
  - Can the interface be customized to the specific needs of a user?
  - Does the interface provide a macro capability that enables a user to identify a sequence of common operations with a single action or command?
- A collection of questions similar to these would be developed for each quality factor to be assessed.

The Software Quality Dilemma

- If you produce a software system that has terrible quality, you lose because no one will want to buy it.
- If on the other hand you spend infinite time, extremely large effort, and huge sums of money to build the absolutely perfect piece of software, then it's going to take so long to complete and it will be so expensive to produce that you'll be out of business anyway.
- Either you missed the market window, or you simply exhausted all your resources.
- So people in industry try to get to that magical middle ground where the product **is good enough** not to be rejected right away, such as during evaluation, but also not the object of so much perfectionism and so much work that it would take too long or cost too much to complete.

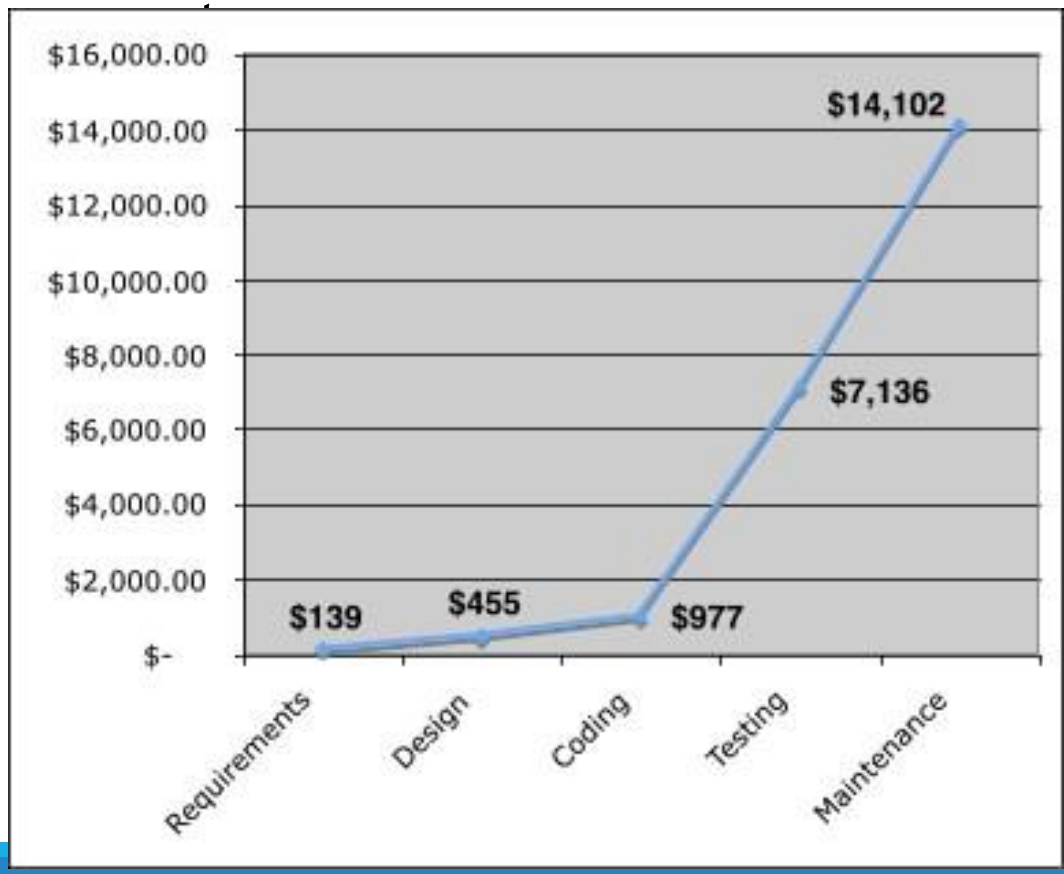
“Good Enough” Software

- Good enough software delivers high quality functions and features that end-users desire, but at the same time it delivers other more obscure or specialized functions and features that contain known bugs.
- Arguments *against* “good enough.”
  - It is true that “good enough” may work in some application domains and for a few major software companies. After all, if a company has a large marketing budget and can convince enough people to buy version 1.0, it has succeeded in locking them in.
  - If you work for a small company be wary of this philosophy. If you deliver a “good enough” (buggy) product, you risk permanent damage to your company’s reputation.
  - You may never get a chance to deliver version 2.0 because bad buzz may cause your sales to plummet and your company to fold.
  - If you work in certain application domains (e.g., real time embedded software, application software that is integrated with hardware can be negligent and open your company to expensive litigation.

Cost of Quality

- *Prevention costs* include
  - quality planning
  - formal technical reviews
  - test equipment
  - Training
- *Internal failure costs* include
  - rework
  - repair
  - failure mode analysis
- *External failure costs* are
  - complaint resolution
  - product return and replacement
  - help line support
  - warranty work

The relative costs to find and repair an error or defect increase dramatically as we go from prevention to detection to internal failure to external failure



Quality and Risk

- *"People bet their jobs, their comforts, their safety, their entertainment, their decisions, and their very lives on computer software. It better be right."*
- Example:
  - *Throughout the month of November, 2000 at a hospital in Panama, 28 patients received massive overdoses of gamma rays during treatment for a variety of cancers. In the months that followed, five of these patients died from radiation poisoning and 15 others developed serious complications. What caused this tragedy? A software package, developed by a U.S. company, was modified by hospital technicians to compute modified doses of radiation for each patient.*
- Poor quality leads to risks, some of them very serious.

Negligence and Liability

- The story is all too common. A governmental or corporate entity hires a major software developer or consulting company to analyze requirements and then design and construct a software-based "system" to support some major activity.
  - The system might support a major corporate function (e.g., pension management) or some governmental function (e.g., healthcare administration or homeland security).
- Work begins with the best of intentions on both sides, but by the time the system is delivered, things have gone bad.
- The system is late, fails to deliver desired features and functions, is error-prone, and does not meet with customer approval.
- Litigation ensues.

Quality and Security

- Gary McGraw comments:

"Software security relates entirely and completely to quality. You must think about security, reliability, availability, dependability—at the beginning, in the design, architecture, test, and coding phases, all through the software life cycle [process]. Even people aware of the software security problem have focused on late life-cycle stuff. The earlier you find the software problem, the better. And there are two kinds of software problems. One is bugs, which are implementation problems. The other is software flaws—architectural problems in the design. People pay too much attention to bugs and not enough on flaws."

- To build a secure system, you must focus on quality, and that focus must begin during design.

The Impact of Management Actions

- As each project task is initiated, a project leader will make decisions that can have a significant impact on product quality.
  - **Estimation decisions**
  - **Scheduling decisions**
  - **Risk-oriented decisions**
- The software quality dilemma can best be summarized by stating Meskimen's law — *There's never time to do it right, but always time to do it over again.*
- *A*dvice: taking the time to do it right is almost never the wrong decision.

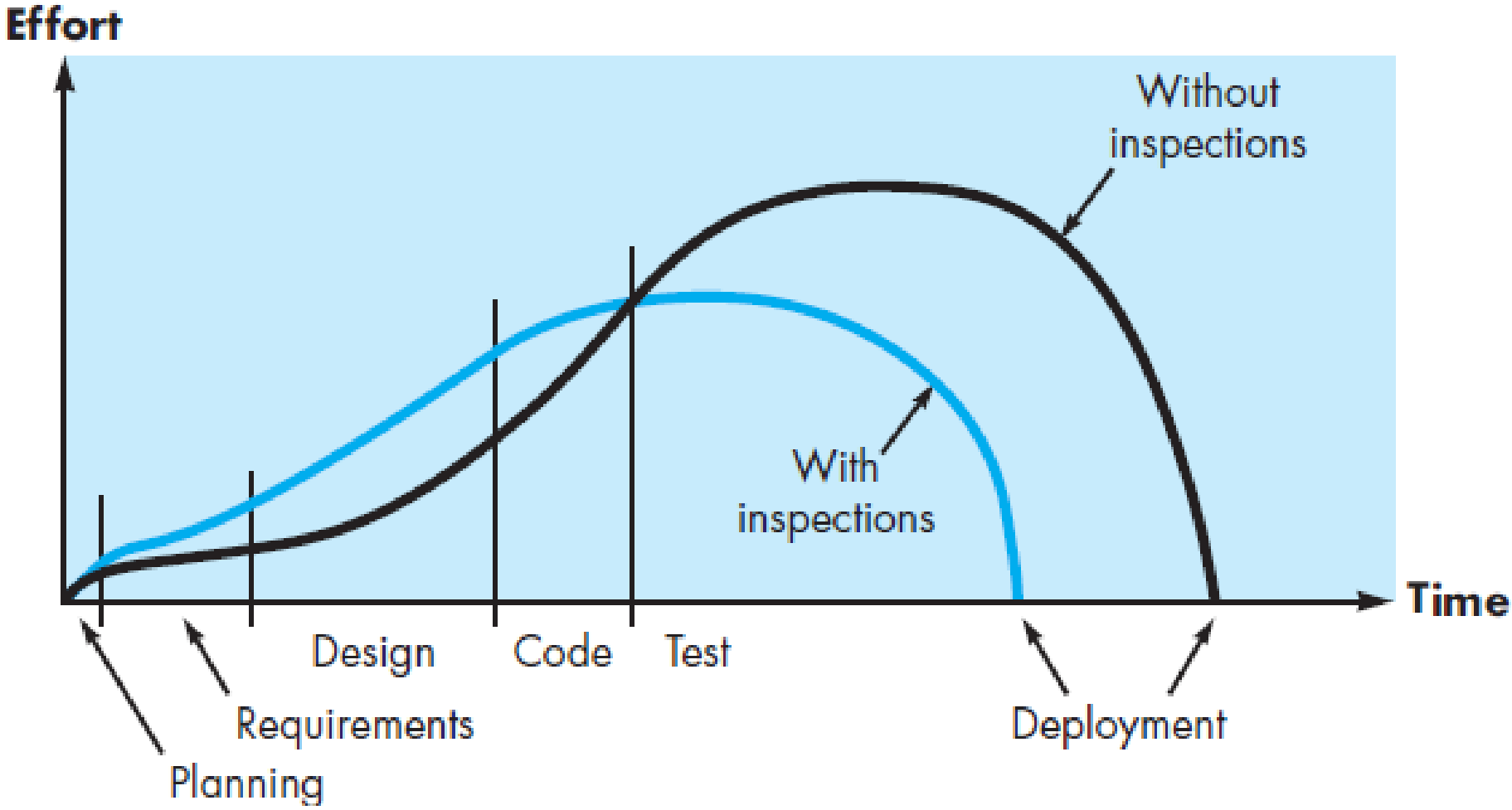
Achieving Software Quality

- Software quality is the result of good project management and solid software engineering practice.
- Management and practice are applied within the context of four broad activities that help a software team achieve high software quality:
  - **Software Engineering Methods:** analysis and design methods that establishes a solid foundation for the construction activity
  - **Project Management Techniques:** if (1) a project manager uses estimation to verify that delivery dates are achievable, (2) schedule dependencies are understood and the team resists the temptation to use shortcuts, (3) risk planning is conducted so problems do not breed chaos, software quality will be affected in a positive way.
  - **Quality Control:** set of software engineering actions that help to ensure that each work product meets its quality goals.
  - **Quality Assurance:** consists of a set of auditing and reporting functions that assess the effectiveness and completeness of quality control actions

Review - software quality control activity

- What Are Reviews?
  - a meeting conducted by technical people for technical people
  - a technical assessment of a work product created during the software engineering process
  - a software quality assurance mechanism
  - a training ground
- What Do We Look For?
  - Errors and defects
    - Error—a quality problem found before the software is released to end users
    - Defect—a quality problem found only after the software has been released to end-users
  - We make this distinction because errors and defects have very different economic, business, psychological, and human impact
  - However, the temporal distinction made between errors and defects in this book is not mainstream thinking

Effort expended with and without reviews



Reference model for technical reviews

Formal Technical Reviews

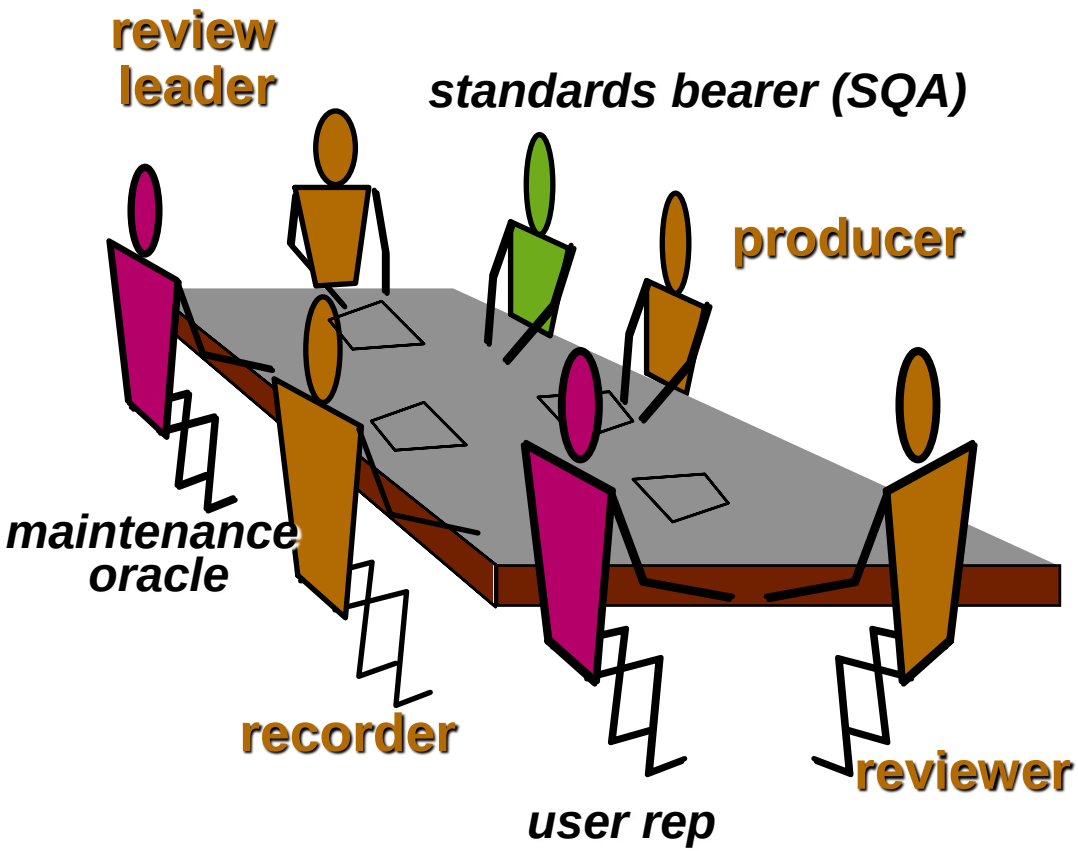
- The objectives of an FTR are:
  - to uncover errors in function, logic, or implementation for any representation of the software
  - to verify that the software under review meets its requirements
  - to ensure that the software has been represented according to predefined standards
  - to achieve software that is developed in a uniform manner
  - to make projects more manageable
- The FTR is actually a class of reviews that includes *walkthroughs* and *inspections*.

The Review Meeting

- Between three and five people (typically) should be involved in the review.
- Advance preparation should occur but should require no more than two hours of work for each person.
- The duration of the review meeting should be less than two hours.
- Focus is on a work product (e.g., a portion of a requirements model, a detailed component design, source code for a component)

The Players

- *Producer*—the individual who has developed the work product
  - informs the project leader that the work product is complete and that a review is required
- *Review leader*—evaluates the product for readiness, generates copies of product materials, and distributes them to two or three *reviewers* for advance preparation.
- *Reviewer(s)*—expected to spend between one and two hours reviewing the product, making notes, and otherwise becoming familiar with the work.
- *Recorder*—reviewer who records (in writing) all important issues raised during the review.



Conducting the Review

- *Review the product, not the producer.*
- *Set an agenda and maintain it.*
- *Limit debate and rebuttal.*
- *Enunciate problem areas, but don't attempt to solve every problem noted.*
- *Take written notes.*
- *Limit the number of participants and insist upon advance 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.*

Review Options Matrix

	IPR*	WT	IN	RRR
trained leader	no	yes	yes	yes
agenda established	maybe	yes	yes	yes
reviewers prepare in advance	maybe	yes	yes	yes
producer presents product	maybe	yes	no	no
"reader" presents product	no	no	yes	no
recorder takes notes	maybe	yes	yes	yes
checklists used to find errors	no	no	yes	no
errors categorized as found	no	no	yes	no
issues list created	no	yes	yes	yes
team must sign-off on result	no	yes	yes	maybe

*IPR—informal peer review WT—Walkthrough
IN—Inspection RRR—round robin review

Sample-Driven Reviews (SDRs)

- SDRs attempt to quantify those work products that are primary targets for full FTRs.

To accomplish this …

- Inspect a fraction a_i of each software work product, *i*. Record the number of faults, f_i found within a_i.
- Develop a gross estimate of the number of faults within work product *i* by multiplying f_i by $1/a_i$.
- Sort the work products in descending order according to the gross estimate of the number of faults in each.
- Focus available review resources on those work products that have the highest estimated number of faults.

Metrics Derived from Reviews

- inspection time per page of documentation
- inspection time per KLOC or FP
- inspection effort per KLOC or FP
- errors uncovered per reviewer hour
- errors uncovered per preparation hour
- errors uncovered per SE task (e.g., design)
- number of minor errors (e.g., typos)
- number of major errors (e.g., nonconformance to req.)
- number of errors found during preparation

Software Project Estimation

- Observations on Estimation
- The Project Planning
- Process Software Project Estimation
- Decomposition Techniques
- Empirical Estimation Models
- Estimation for Object-Oriented Projects

Software Project Estimation

- Software project management begins with a set of activities that are collectively called project planning
- Software project planning encompasses five major activities—estimation, scheduling, risk analysis, quality management planning, and change management planning

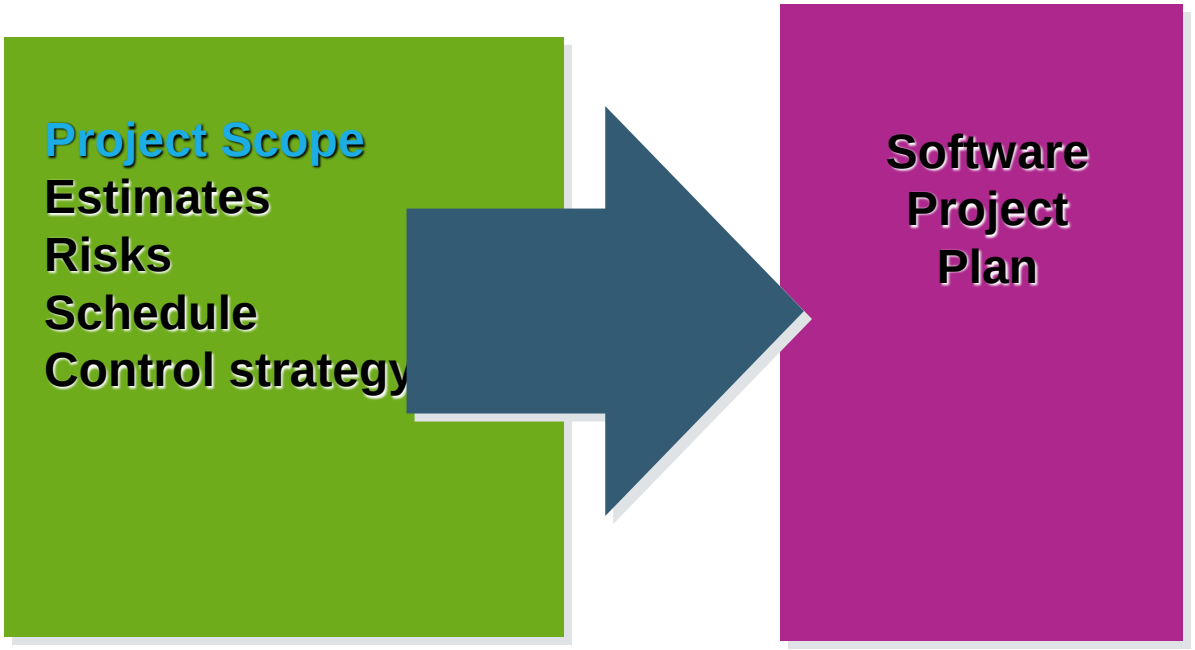
Observations on estimation

- Whenever estimates are made, you look into the future and accept some degree of uncertainty as a matter of course
- Estimation lays a foundation for all other project planning actions, and project planning provides the road map for successful software engineering
- the software team should
  - estimate the work to be done
  - the resources that will be required
  - the time that will elapse from start to finish.
- Estimation requires
  - experience,
  - access to good historical information (metrics)
  - the courage to commit to quantitative predictions when qualitative information is all that exists.
- Estimation carries inherent risk, and this risk leads to uncertainty.

Observations on estimation

- factor that can affect the accuracy, reliability and efficacy of estimates
  - Project complexity
  - Project size
  - the degree of structural uncertainty
- If project scope is poorly understood or project requirements are subject to change, uncertainty and estimation risk become dangerously high.

Write it Down!



41

Software Project Planning

- provide a framework that enables the manager to make reasonable estimates of resources, cost, and schedule.
- The overall goal of project planning is to establish a pragmatic strategy for controlling, tracking, and monitoring a complex technical project.

Why?

- *So the end result gets done on time, with quality!*

Project Planning Task Set-I

- Establish project scope
- Determine feasibility
- Analyze risks
  - Risk analysis.
- Define required resources
  - Determine require human resources
  - Define reusable software resources
  - Identify environmental resources

43

Project Planning Task Set-II

- Estimate cost and effort
  - Decompose the problem
  - Develop two or more estimates using size, function points, process tasks or use-cases
  - Reconcile the estimates
- Develop a project schedule
  - Scheduling.
    - Establish a meaningful task set
    - Define a task network
    - Use scheduling tools to develop a timeline chart
    - Define schedule tracking mechanisms

44

Software Project Estimation

- Software cost and effort estimation - never be an exact science.
- Too many variables—human, technical, environmental, political—can affect the ultimate cost of software and effort applied to develop it.
- However, software project estimation can be transformed from a black art to a series of systematic steps that provide estimates with acceptable risk.

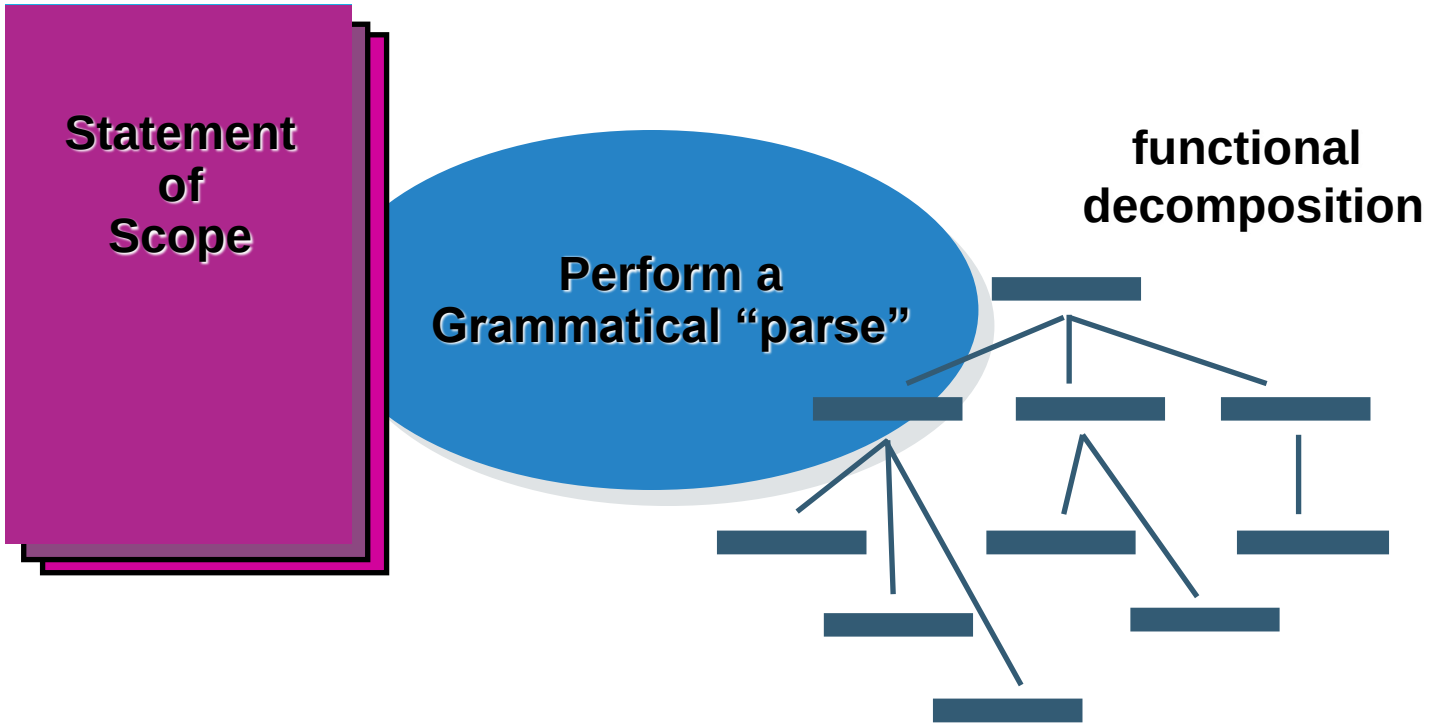
Options for reliable cost and effort estimates -Estimation Techniques

- Delay estimation until late in the project - not practical
- Base estimates on similar projects that have already been completed - work reasonably well, if the current project is quite similar to past efforts
- Use relatively simple decomposition techniques to generate project cost and effort estimates - stepwise
- Use one or more empirical models for software cost and effort estimation -complement decomposition techniques
- Automated tools based on decomposition techniques or empirical models and provide an attractive option for estimation

Decomposition techniques

- the decomposition approach -two different points of view:
  - decomposition of the problem
  - decomposition of the process
- Prerequisite for decomposition:
  - scope of the software to be built
  - generate an estimate of its “size.”

Functional Decomposition



48

Estimation Accuracy

- predicated on a number of things:
  (1) the degree to which you have properly estimated the size of the product to be built;
  (2) the ability to translate the size estimate into human effort, calendar time, and dollars (a function of the availability of reliable software metrics from past projects);
  (3) the degree to which the project plan reflects the abilities of the software team;
  (4) the stability of product requirements and the environment that supports the software engineering effort.

Sizing software

- software sizing represents first major challenge as a planner
- Size refers to a quantifiable outcome of the software project
  - direct approach to measure size - lines of code (LOC)
  - indirect approach to measure size - function points (FP)
- Size can be estimated by considering
  - the type of project and its application domain,
  - the functionality delivered (i.e., the number of function points),
  - the number of components to be delivered,
  - the degree to which a set of existing components must be modified for the new system

Problem-Based Estimation

- LOC and FP data are used in two ways during software project estimation:

(1) as estimation variables to “size” each element of the software and

(2) as baseline metrics collected from past projects and used in conjunction with estimation variables to develop cost and effort projections.

Problem-Based Estimation

Common characteristics LOC and FP estimation

- Begin with a bounded statement of software scope and attempt to decompose the statement of scope into problem functions that can each be estimated individually.
- Estimate LOC or FP (the estimation variable) for each function.
  - Alternatively, you may choose another component for sizing, such as classes or objects, changes, or business processes affected.
- Baseline productivity metrics (e.g., LOC/pm or FP/pm) are then applied to the appropriate estimation variable, and cost or effort for the function is derived.
- Function estimates are combined to produce an overall estimate for the entire project.

Problem-Based Estimation

Different characteristics LOC and FP estimation

- The level of detail required for decomposition and the target of the partitioning.
  - LOC- decomposition is essential- greater the degree of partitioning, the more likely reasonably accurate estimates of LOC can be developed
  - FP - Rather than focusing on function, each of the information domain characteristics—inputs, outputs, data files, inquiries, and external interfaces—as well as the complexity adjustment values are estimated., which can be used to derive an FP value that can be tied to past data and used to generate an estimate.

Problem-Based Estimation

- Begin by estimating a range of values for each function or information domain value Irrespective of LOC/FP variable
- Using historical data or (when all else fails) intuition, estimate an optimistic, most likely, and pessimistic size value for each function or count for each information domain value.
- An implicit indication of the degree of uncertainty is provided when a range of values is specified.
- The expected value for the estimation variable (size) S can be computed as a weighted average of
  - the optimistic (s_{opt})
  - most likely (s_m)
  - pessimistic (s_{pess}) estimates.

Problem-Based Estimation -example

- a software package to be developed for a computer-aided design application for mechanical components. The software is to execute on a desktop workstation and must interface with various computer graphics peripherals including a mouse, digitizer, high-resolution color display, and laser printer.
- A preliminary statement of software scope can be developed: The mechanical CAD software will accept two- and three-dimensional geometric data from a designer. The designer will interact and control the CAD system through a user interface that will exhibit characteristics of good human/machine interface design. All geometric data and other supporting information will be maintained in a CAD database. Design analysis modules will be developed to produce the required output, which will be displayed on a variety of devices. The software will be designed to control and interact with peripheral devices that include a mouse, scanner, laser printer, and plotter.

Example: LOC Approach

Function	Estimated LOC
user interface and control facilities (UICF)	2,300
two-dimensional geometric analysis (2DGA)	5,300
three-dimensional geometric analysis (3DGA)	6,800
database management (DBM)	3,350
computer graphics display facilities (CGDF)	4,950
peripheral control (PC)	2,100
design analysis modules (DAM)	8,400
estimated lines of code	33,200

The range of LOC estimates for the 3D geometric analysis function

- optimistic, 4600 LOC;
- most likely, 6900 LOC;
- pessimistic, 8600 LOC.
- The expected value is (4600+6900+8600)/3=6800 LOC.

A review of historical data indicates organizational : Average productivity for systems of this type = 620 LOC/pm.

Burdened labor rate =$8000 per month, the cost per line of code is approximately $13.

Based on the LOC estimate and the historical productivity data, the total estimated project cost is **$431,000 and the estimated effort is 54 person-months.**

56

Example: FP Approach

Information Domain Value	opt.	likely	pess.	est. count	weight	FP-count
number of inputs	20	24	30	24	4	97
number of outputs	12	15	22	16	5	78
number of inquiries	16	22	28	22	5	88
number of files	4	4	5	4	10	42
number of external interfaces	2	2	3	2	7	15
count-total						**321**

Factor	Value
Backup and recovery	4
Data communications	2
Distributed processing	0
Performance critical	4
Existing operating environment	3
Online data entry	4
Input transaction over multiple screens	5
Master files updated online	3
Information domain values complex	5
Internal processing complex	5
Code designed for reuse	4
Conversion/installation in design	3
Multiple installations	5
Application designed for change	5
Value adjustment factor	**1.17**

Sum(Fi) = 52

The estimated number of FP is derived:

$$FP_{estimated} = \text{count-total} \times [0.65 + 0.01 \times \text{Sum}(F_i)]$$

$$FP_{estimated} = 375$$

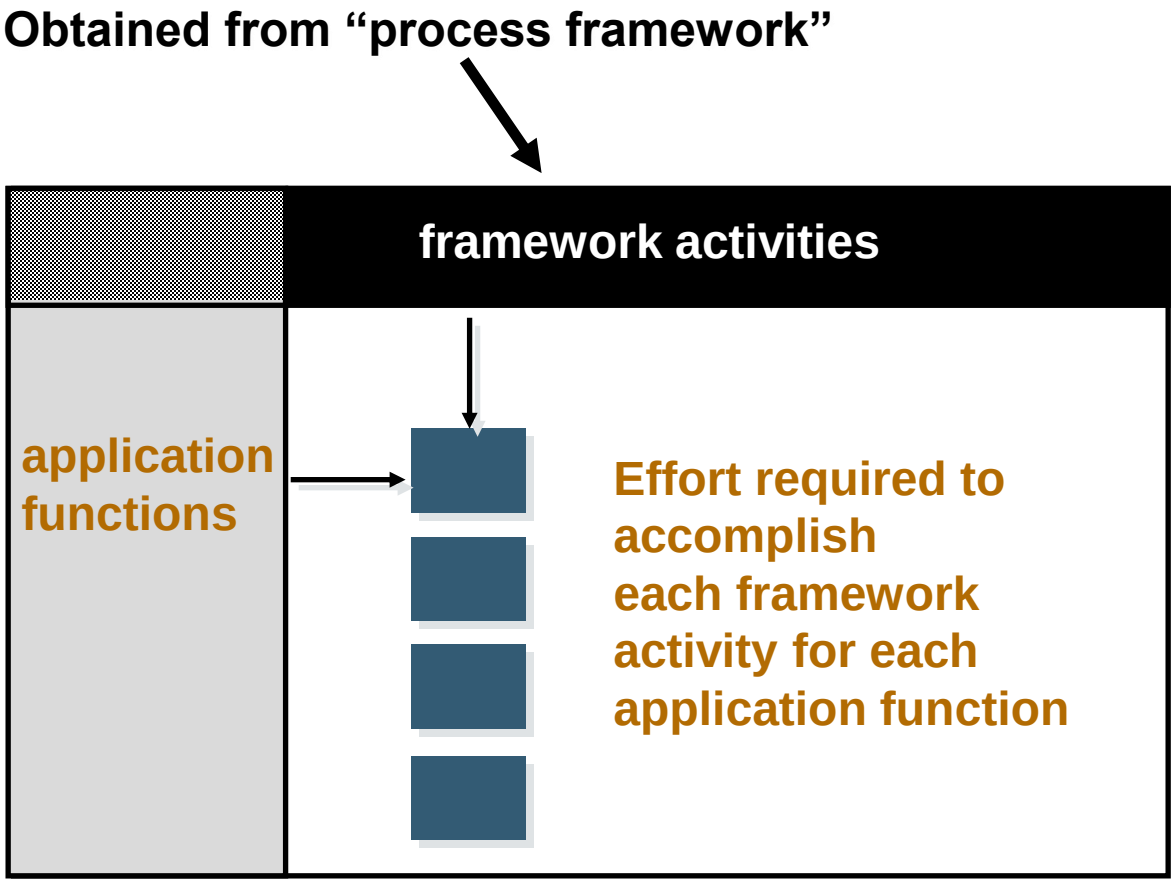
Organizational average productivity = 6.5 FP/pm.

burdened labor rate = $8000 per month, approximately $1230/FP.

Based on the FP estimate and the historical productivity data, **total estimated project cost is 375* 1230 = $461,000 and estimated effort is (375/6.5 FP/pm) =58 person-months.**

57

Process-Based Estimation



Consider the same CAD software development project.

58

Process-Based Estimation Example

Activity →	CC	Planning	Risk Analysis	Engineering		Construction Release		CE	Totals
Task →				analysis	design	code	test		
Function									
▼									
UICF				0.50	2.50	0.40	5.00	n/a	8.40
2DGA				0.75	4.00	0.60	2.00	n/a	7.35
3DGA				0.50	4.00	1.00	3.00	n/a	8.50
CGDF				0.50	3.00	1.00	1.50	n/a	6.00
DSM				0.50	3.00	0.75	1.50	n/a	5.75
PCF				0.25	2.00	0.50	1.50	n/a	4.25
DAM				0.50	2.00	0.50	2.00	n/a	5.00
Totals	0.25	0.25	0.25	3.50	20.50	4.50	16.50		46.00
% effort	1%	1%	1%	8%	45%	10%	36%		

CC = customer communication CE = customer evaluation

Based on an average burdened labor rate of $8,000 per month, **the total estimated project cost is $368,000 and the estimated effort is 46 person-months.**

59

Estimation with Use-Cases

- Constraints about Use cases
  - Use cases are described using many different formats and styles and represent an external view (the user's view) of the software.
    - They can be written at many different levels of abstraction.
    - Use cases do not address the complexity of the functions and features that are described, and they can describe complex behavior (e.g., interactions) that involve many functions and features.
- Even with these constraints, it is possible to compute use case points (UCPs) in a manner that is analogous to the computation of functions points
- computation of use case points must take the following characteristics into account:
  - The number and complexity of the use cases in the system.
  - The number and complexity of the actors on the system.
  - Various nonfunctional requirements (such as portability, performance, maintainability) that are not written as use cases.
  - The environment in which the project will be developed (e.g., the programming language, the software team's motivation).

Use case complexity assessment

- Each use case is assessed to determine its relative complexity.
  - A simple use case indicates a simple user interface, a single database, and three or fewer transactions and five or fewer class implementations.
  - An average use case indicates a more complex UI, 2 or 3 databases, and 4 to 7 transactions with 5 to 10 classes.
  - A complex use case implies a complex UI with multiple databases, using eight or more transactions and 11 or more classes.
- Each use case is assessed using these criteria and the count of each type is weighted by a factor of 5, 10, and 15, respectively.
- A total unadjusted use case weight (UUCW) is the sum of all weighted counts.

Use case Actor assessment

- Simple actors are automatons (another system, a machine or device) that communicate through an API.
- Average actors are automatons that communicate through a protocol or a data store,
- Complex actors are humans who communicate through a GUI or other human interface.
- The count of each type actor is weighted by a factor of 1, 2, and 3, respectively.
- The total unadjusted actor weight (UAW) is the sum of all weighted counts.

Estimation with Use-Cases Example

- These unadjusted values are modified by considering thirteen technical complexity factors (TCFs) and eight environment complexity factors (ECFs).
- Once these values have been determined, the final UCP value is computed in the following manner:

$$UCP = (UUCW + UAW) \times TCF \times ECF$$

Three subsystem in CAD software	Use cases	actors
User interface	16 complex	8 complex 12 average 4 simple
Engineering	14 average and 8 simple	
Infrastructure	10 simple	

UUCW = (16 use cases x 15)+[(14 use cases x 10) + (8 use cases x 5)] + (10 use cases x 5) = 470

UAW = (8 actors x 1) 1 (12 actors x 2) + 4 actors x 3) = 44

After evaluation of the technology and the environment, TCF = 1.04 and ECF = 0.96

UCP = (470 + 44) x 1.04 x 0.96 = 513

Estimation with Use-Cases Example

- Using past project data as a guide, the development group has produced 85 LOC per UCP.
  - An estimate of the overall size of the CAD project is 43,600 LOC.
- Using 620 LOC/pm as the average productivity for systems of this type and a burdened labor rate of $8,000 per month, the cost per line of code is approximately $13.
  - Based on the use-case estimate and the historical productivity data, the total estimated project cost is $552,000 and the estimated effort is about 70 person-months

Empirical Estimation Models

- uses empirically derived formulas to predict effort as a function of LOC or FP.
- Empirical data - derived from a limited sample of projects
  - no estimation model is appropriate for all classes of software and in all development environments.
  - should use the results obtained from such models judiciously.
- The model should be tested by applying data collected from completed projects, plugging the data into the model, and then comparing actual to predicted results.
  - If agreement is poor, the model must be tuned and retested before it can be used.

Structure of Estimation Models

- derived using regression analysis on data collected from past software projects.
- The overall structure of such models takes the form

$$E = A + B \times (e_v)^C$$

  - where A, B, and C are empirically derived constants,
  - E is effort in person-months,
  - e_v is the estimation variable (either LOC or FP).
- The majority of estimation models have some form of project adjustment component that enables E to be adjusted by other project characteristics (e.g., problem complexity, staff experience, development environment).
- A quick examination of any empirically derived model indicates that it must be calibrated for local needs.

COnstructive COst Model (COCOMO)

- COCOMO II is actually a hierarchy of estimation models that address the following areas:
  - *Application composition model.* Used during the early stages of software engineering, when prototyping of user interfaces, consideration of software and system interaction, assessment of performance, and evaluation of technology maturity are paramount.
  - *Early design stage model.* Used once requirements have been stabilized and basic software architecture has been established.
  - *Post-architecture-stage model.* Used during the construction of the software.

67

COCOMO-II

- The software equation is a dynamic multivariable model that assumes a specific distribution of effort over the life of a software development project.
- The model has been derived from productivity data collected for over 4,000 contemporary software 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 skills factor"
P = "productivity parameter" - derived for local conditions using historical data collected from past development efforts. It reflects:

- overall process maturity and management practices,
- the extent to which good software engineering practices are used
- the level of programming languages used
- the state of the software environment
- the skills and experience of the software team
- the complexity of the application

COCOMO-II

- Minimum development time is defined as t_{min} $=8.14(LOC/P^{0.43})$ in months for t_{min} >6 months
- E = 180 Bt^3 in person-months for E >= 20 person-months and t is represented in years.
- Typical values of P might be
  - 2,000 for development of real-time embedded software
  - 10,000 for telecommunication and systems software,
  - 28,000 for business systems application
- for the CAD software with P = 12,000
  - t_{min} = 8.14 $(33{,}200/12{,}000^{0.43})$= 12.6 calendar months
  - E = 180 x 0.28 x $(1.05)^3$ = 58 person-months

Estimation for OO Projects

1. Develop estimates using effort decomposition, FP analysis, and any other method that is applicable for conventional applications.
2. Using the requirements model, develop use cases and determine a count. Recognize that the number of use cases may change as the project progresses.
3. From the requirements model, determine the number of key classes (called analysis classes).
4. Categorize the type of interface for the application and develop a multiplier for support classes. Multiply the number of key classes (step 3) by the multiplier to obtain an estimate for the number of support classes.

Interface type	Multiplier
No GUI	2.0
a text-based user interface	2.25
Conventional GUI	2.5
Complex GUI	3.0

5. Multiply the total number of classes (key 1 support) by the average number of work units per class it is suggested 15 to 20 person-days per class.
6. Cross-check the class-based estimate by multiplying the average number of work units per use case.

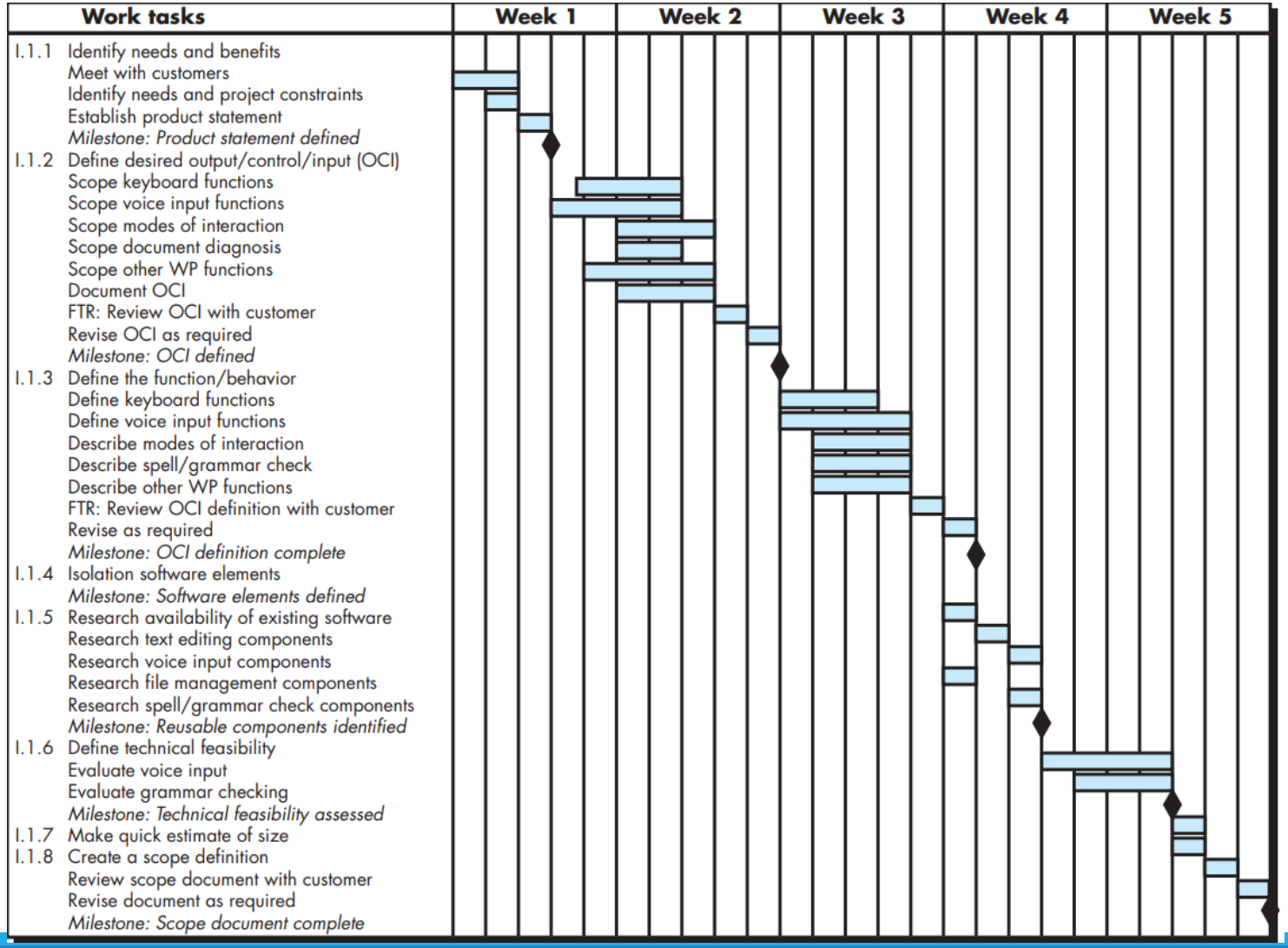
Project scheduling

- Establish a project schedule that defines
  - software engineering tasks and milestones
  - identifies who is responsible for conducting each task
  - specifies the intertask dependencies that may have a strong bearing on progress
- Program evaluation and review technique (PERT) and the critical path method (CPM) are two project scheduling methods that can be applied to software development
- Driven by project planning activities: estimates of effort, a decomposition of the product function, the selection of the appropriate process model and task set, and decomposition of the tasks that are selected
- Tasks, sometimes called the project work breakdown structure (WBS), are defined for the product as a whole or for individual functions.

Time-Line Charts

- PERT and CPM provide quantitative tools that allow you to
  - (1) determine the critical path—the chain of tasks that determines the duration of the project,
  - (2) establish “most likely” time estimates for individual tasks by applying statistical models, and
  - (3) calculate “boundary times” that define a time “window” for a particular task.
- Effort, duration, and start date are then input for each task.
  - a time-line chart, also called a Gantt chart, is generated
- Time line chart example - concept scoping task for a word-processing (WP) software product.
  - All project tasks (for concept scoping) are listed in the left-hand column.
  - The horizontal bars indicate the duration of each task.
  - When multiple bars occur at the same time on the calendar, task concurrency is implied.
  - The diamonds indicate milestones.

Time-Line Charts



Time-Line Charts

- Project table : a tabular listing of all project tasks is produced-
  - their planned and actual start and end dates, and a variety of related information

Work tasks	Planned start	Actual start	Planned complete	Actual complete	Assigned person	Effort allocated	Notes
I.1.1 Identify needs and benefits							Scoping will require more effort/time
Meet with customers	wk1, d1	wk1, d1	wk1, d2	wk1, d2	BLS	2 p-d	
Identify needs and project constraints	wk1, d2	wk1, d2	wk1, d2	wk1, d2	JPP	1 p-d	
Establish product statement	wk1, d3	wk1, d3	wk1, d3	wk1, d3	BLS/JPP	1 p-d	
Milestone: Product statement defined	wk1, d3	wk1, d3	wk1, d3	wk1, d3			
I.1.2 Define desired output/control/input (OCI)							
Scope keyboard functions	wk1, d4	wk1, d4	wk2, d2		BLS	1.5 p-d	
Scope voice input functions	wk1, d3	wk1, d3	wk2, d2		JPP	2 p-d	
Scope modes of interaction	wk2, d1		wk2, d3		MLL	1 p-d	
Scope document diagnostics	wk2, d1		wk2, d2		BLS	1.5 p-d	
Scope other WP functions	wk1, d4	wk1, d4	wk2, d3		JPP	2 p-d	
Document OCI	wk2, d1		wk2, d3		MLL	3 p-d	
FTR: Review OCI with customer	wk2, d3		wk2, d3		all	3 p-d	
Revise OCI as required	wk2, d4		wk2, d4		all	3 p-d	
Milestone: OCI defined	wk2, d5		wk2, d5				
I.1.3 Define the function/behavior							

Tracking the Schedule

- The project schedule becomes a road map that defines the tasks and milestones to be tracked and controlled as the project proceeds.
- Tracking can be accomplished in a number of different ways:
  - Conducting periodic project status meetings in which each team member reports progress and problems.
  - Evaluating the results of all reviews conducted throughout the software engineering process.
  - Determining whether formal project milestones (the diamonds) have been accomplished by the scheduled date.
  - Comparing the actual start date to the planned start date for each project task listed in the resource table
  - Meeting informally with practitioners to obtain their subjective assessment of progress to date and problems on the horizon.
  - Using earned value analysis to assess progress quantitatively.

Tracking Progress for an OO Project

- Task parallelism in iterative model makes project tracking difficult
- Difficulty for establishing meaningful milestones for an OO project because a number of different things are happening at once.
- Recalling that the OO process model is iterative, each of these milestones may be revisited as different increments are delivered to the customer

Scheduling for WebApp/Mobile Projects

- During the first iteration, a macroscopic schedule is developed.
  - identifies all Web or MobileApp increments and projects the dates on which each will be deployed
- As the development of an increment gets under way, the entry for the increment on the macroscopic schedule is refined into a detailed schedule.
  - Deployment dates (represented by diamonds on the time-line chart) are preliminary and may change as more detailed scheduling of the increments occurs.

Major milestones -considered completed

- Technical milestone: OO analysis completed
  - All classes and the class hierarchy have been defi ned and reviewed.
  - Class attributes and operations associated with a class have been defi ned and reviewed.
  - Class relationships (Chapter 10) have been established and reviewed.
  - A behavioral model (Chapter 11) has been created and reviewed.
  - Reusable classes have been noted.
- Technical milestone: OO design completed
  - The set of subsystems has been defi ned and reviewed.
  - Classes are allocated to subsystems and reviewed.
  - Task allocation has been established and reviewed.
  - Responsibilities and collaborations have been identified.
  - Attributes and operations have been designed and reviewed.
  - The communication model has been created and reviewed.
- Technical milestone: OO programming completed
  - Each new class has been implemented in code from the design model.
  - Extracted classes (from a reuse library) have been implemented.
  - Prototype or increment has been built.
- Technical milestone: OO testing
  - The correctness and completeness of OO analysis and design models have been reviewed.
  - A class-responsibility-collaboration network (Chapter 10) has been developed and reviewed.
  - Test cases are designed, and class-level tests (Chapter 24) have been conducted for each class.
  - Test cases are designed, and cluster testing (Chapter 24) is completed and the classes are integrated.
  - System-level tests have been completed.

Risk management

- The process by which a course of action is selected that balances the potential impact of a risk weighted by its probability of occurrence and the benefits of avoiding (or controlling) the risk
- Risk management: Reactive vs Proactive

Reactive versus proactive risk strategies

- Reactive Risk Management
  - project team reacts to risks when they occur
  - mitigation—plan for additional resources in anticipation of fire fighting
  - fix on failure—resource are found and applied when the risk strikes
  - crisis management—failure does not respond to applied resources and project is in jeopardy
- Proactive Risk Management
  - formal risk analysis is performed
  - organization corrects the root causes of risk
    - TQM concepts and statistical SQA
    - examining risk sources that lie beyond the bounds of the software
    - developing the skill to manage change

Software risks

- A risk is a potential problem – it might happen and it might not
- Conceptual definition of risk
  - Risk concerns future happenings
  - Risk involves change in mind, opinion, actions, places, etc.
  - Risk involves choice and the uncertainty that choice entails
- Two characteristics of risk
  - Uncertainty – the risk may or may not happen, that is, there are no 100% risks (those, instead, are called constraints)
  - Loss – the risk becomes a reality and unwanted consequences or losses occur

Software risks Categorization – approach 1

- **Project risks**
  - They threaten the project plan
  - If they become real, it is likely that the project schedule will slip and that costs will increase
- **Technical risks**
  - They threaten the quality and timeliness of the software to be produced
  - If they become real, implementation may become difficult or impossible
- **Business risks**
  - They threaten the viability of the software to be built
  - If they become real, they jeopardize the project or the product

Software risks Categorization – approach 1

- Sub-categories of Business risks
  - **Market risk** – building an excellent product or system that no one really wants
  - **Strategic risk** – building a product that no longer fits into the overall business strategy for the company
  - **Sales risk** – building a product that the sales force doesn't understand how to sell
  - **Management risk** – losing the support of senior management due to a change in focus or a change in people
  - **Budget risk** – losing budgetary or personnel commitment

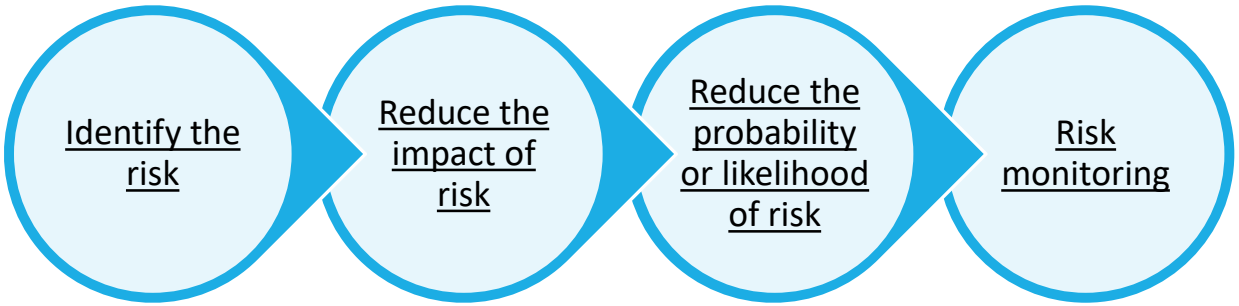
Software risks Categorization – approach 2

- Known risks
  - Those risks that can be uncovered after careful evaluation of the project plan, the business and technical environment in which the project is being developed, and other reliable information sources (e.g., unrealistic delivery date)
- Predictable risks
  - Those risks that are extrapolated from past project experience (e.g., past turnover)
- Unpredictable risks
  - Those risks that can and do occur, but are extremely difficult to identify in advance

Risk Management

- Risk management is carried out to:
  - Identify the risk
  - Reduce the impact of risk
  - Reduce the probability or likelihood of risk
  - Risk monitoring

1) Identify possible risks; recognize what can go wrong
2) Analyze each risk to estimate the probability that it will occur and the impact (i.e., damage) that it will do if it does occur
3) Rank the risks by probability and impact
   - Impact may be negligible, marginal, critical, and catastrophic
4) Develop a contingency plan to manage those risks having high probability and high impact



85

Seven Principles

- Maintain a global perspective—view software risks within the context of system and the business problem
- Take a forward-looking view—think about the risks that may arise in the future; establish contingency plans
- Encourage open communication—if someone states a potential risk, don't discount it.
- Integrate—a consideration of risk must be integrated into the software process
- Emphasize a continuous process—the team must be vigilant throughout the software process, modifying identified risks as more information is known and adding new ones as better insight is achieved.
- Develop a shared product vision—if all stakeholders share the same vision of the software, it likely that better risk identification and assessment will occur.
- Encourage teamwork—the talents, skills and knowledge of all stakeholder should be pooled

Risk Identification -Background

- Risk identification is a systematic attempt to specify threats to the project plan
- By identifying known and predictable risks, the project manager takes a first step toward avoiding them when possible and controlling them when necessary
- Generic risks
  - Risks that are a potential threat to every software project
- Product-specific risks
  - Risks that can be identified only by those with a clear understanding of the technology, the people, and the environment that is specific to the software that is to be built
  - This requires examination of the project plan and the statement of scope
  - "What special characteristics of this product may threaten our project plan?"

87

Risk Item Checklist

Used as one way to identify risks

Focuses on known and predictable risks in specific subcategories

Can be organized in several ways

- A list of characteristics relevant to each risk subcategory
- Questionnaire that leads to an estimate on the impact of each risk
- A list containing a set of risk component and drivers and their probability of occurrence

88

Known and Predictable Risk Categories

1-Product size	risks associated with overall size of the software to be built
2-Business impact	risks associated with constraints imposed by management or the marketplace
3-Customer characteristics	risks associated with sophistication of the customer and the developer's ability to communicate with the customer in a timely manner
4-Process definition	risks associated with the degree to which the software process has been defined and is followed
5-Development environment	risks associated with availability and quality of the tools to be used to build the project
6-Technology to be built	risks associated with complexity of the system to be built and the "newness" of the technology in the system
7-Staff size and experience	risks associated with overall technical and project experience of the software engineers who will do the work

Recording Risk Information

Project: Embedded software for XYZ system
Risk type: schedule risk
Priority (1 low ... 5 critical): 4
Risk factor: Project completion will depend on tests which require hardware component under development. Hardware component delivery may be delayed
Probability: 60 %
Impact: Project completion will be delayed for each day that hardware is unavailable for use in software testing
Monitoring approach: Scheduled milestone reviews with hardware group
Contingency plan: Modification of testing strategy to accommodate delay using software simulation
Estimated resources: 6 additional person months beginning 7-1-96

Questionnaire on Project Risk

(Questions are ordered by their relative importance to project success)

1) Have top software and customer managers formally committed to support the project?
2) Are end-users enthusiastically committed to the project and the system/product to be built?
3) Are requirements fully understood by the software engineering team and its customers?
4) Have customers been involved fully in the definition of requirements?
5) Do end-users have realistic expectations?
6) Is the project scope stable?

(More on next slide)

Questionnaire on Project Risk (continued)

7) Does the software engineering team have the right mix of skills?

8) Are project requirements stable?

9) Does the project team have experience with the technology to be implemented?

10) Is the number of people on the project team adequate to do the job?

11) Do all customer/user constituencies agree on the importance of the project and on the requirements for the system/product to be built?

92

Risk Components and Drivers

- The project manager identifies the risk drivers that affect the following risk components
  - **Performance risk** - the degree of uncertainty that the product will meet its requirements and be fit for its intended use
  - **Cost risk** - the degree of uncertainty that the project budget will be maintained
  - **Support risk** - the degree of uncertainty that the resultant software will be easy to correct, adapt, and enhance
  - **Schedule risk** - the degree of uncertainty that the project schedule will be maintained and that the product will be delivered on time
- The impact of each risk driver on the risk component is divided into one of four impact levels
  - Negligible, marginal, critical, and catastrophic
- Risk drivers can be assessed as impossible, improbable, probable, and frequent

93

Risk Projection (Estimation)- Background

- Risk projection (or estimation) attempts to rate each risk in two ways
  - The probability that the risk is real
  - The consequence of the problems associated with the risk, should it occur
- The project planner, managers, and technical staff perform four risk projection steps
- The intent of these steps is to consider risks in a manner that leads to prioritization
- By prioritizing risks, the software team can allocate limited resources where they will have the most impact

94

Risk Projection/Estimation Steps

1) Establish a scale that reflects the perceived likelihood of a risk (e.g., 1-low, 10-high)
2) Delineate the consequences of the risk
3) Estimate the impact of the risk on the project and product
4) Note the overall accuracy of the risk projection so that there will be no misunderstandings

95

Contents of a Risk Table

- A risk table provides a project manager with a simple technique for risk projection
- It consists of five columns
  - Risk Summary – short description of the risk
  - Risk Category – one of seven risk categories
  - Probability – estimation of risk occurrence based on group input
  - Impact – (1) catastrophic (2) critical (3) marginal (4) negligible
  - RMMM –the Risk Mitigation, Monitoring, and Management Plan

Risk Summary	Risk Category	Probability	Impact (1-4)	RMMM

96

Developing a Risk Table

- List all risks in the first column (by way of the help of the risk item checklists)
- Mark the category of each risk
- Estimate the probability of each risk occurring
- Assess the impact of each risk based on an averaging of the four risk components to determine an overall impact value
- Sort the rows by probability and impact in descending order
- Draw a horizontal cutoff line in the table that indicates the risks that will be given further attention

97

Assessing Risk Impact

- Three factors affect the consequences that are likely if a risk does occur
  - **Its nature** – This indicates the problems that are likely if the risk occurs
  - **Its scope** – This combines the severity of the risk (how serious was it) with its overall distribution (how much was affected)
  - **Its timing** – This considers when and for how long the impact will be felt
- The overall risk exposure formula is RE = P x C
  - P = the probability of occurrence for a risk
  - C = the cost to the project should the risk actually occur
- Example
  - P = 80% probability that 18 of 60 software components will have to be developed
  - C = Total cost of developing 18 components is $25,000
  - RE = .80 x $25,000 = $20,000

98

Risk refinement

- represent the risk in condition-transition-consequence
- General condition can be refined in the following manner:
  - Subcondition 1. Certain reusable components were developed by a third party with no knowledge of internal design standards.
  - Subcondition 2. The design standard for component interfaces has not been solidified and may not conform to certain existing reusable components.
  - Subcondition 3. Certain reusable components have been implemented in a language that is not supported on the target environment.
- The consequences associated with these refined subconditions remain the
- but the refinement helps to isolate the underlying risks and might lead to easier analysis and response.

Risk Mitigation, Monitoring, and Management

Background

- An effective strategy for dealing with risk must consider three issues

  (Note: these are not mutually exclusive)

  - Risk mitigation (i.e., avoidance)
  - Risk monitoring
  - Risk management and contingency planning

- Risk mitigation (avoidance) is the primary strategy and is achieved through a plan
  - Example: Risk of high staff turnover

(More on next slide)

Background (continued)

Strategy for Reducing Staff Turnover

- 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 ensure that documents are developed in a timely manner
- Conduct peer reviews of all work (so that more than one person is "up to speed")
- Assign a backup staff member for every critical technologist

102

Background (continued)

- During risk monitoring, the project manager monitors factors that may provide an indication of whether a risk is becoming more or less likely
- Risk management and contingency planning assume that mitigation efforts have failed and that the risk has become a reality
- RMMM steps incur additional project cost
  - Large projects may have identified 30 – 40 risks
- Risk is not limited to the software project itself
  - Risks can occur after the software has been delivered to the user

103

Background (continued)

- Software safety and hazard analysis
  - These are software quality assurance activities that focus on the identification and assessment of potential hazards that may affect software negatively and cause an entire system to fail
  - If hazards can be identified early in the software process, software design features can be specified that will either eliminate or control potential hazards

104

The RMMM Plan

- The RMMM plan may be a part of the software development plan or may be a separate document
- Once RMMM has been documented and the project has begun, the risk mitigation, and monitoring steps begin
  - Risk mitigation is a problem avoidance activity
  - Risk monitoring is a project tracking activity
- Risk monitoring has three objectives
  - To assess whether predicted risks do, in fact, occur
  - To ensure that risk aversion steps defined for the risk are being properly applied
  - To collect information that can be used for future risk analysis
- The findings from risk monitoring may allow the project manager to ascertain what risks caused which problems throughout the project

105

Summary

- Whenever much is riding on a software project, common sense dictates risk analysis
  - Yet, most project managers do it informally and superficially, if at all
- However, the time spent in risk management results in
  - Less upheaval during the project
  - A greater ability to track and control a project
  - The confidence that comes with planning for problems before they occur
- Risk management can absorb a significant amount of the project planning effort…but the effort is worth it

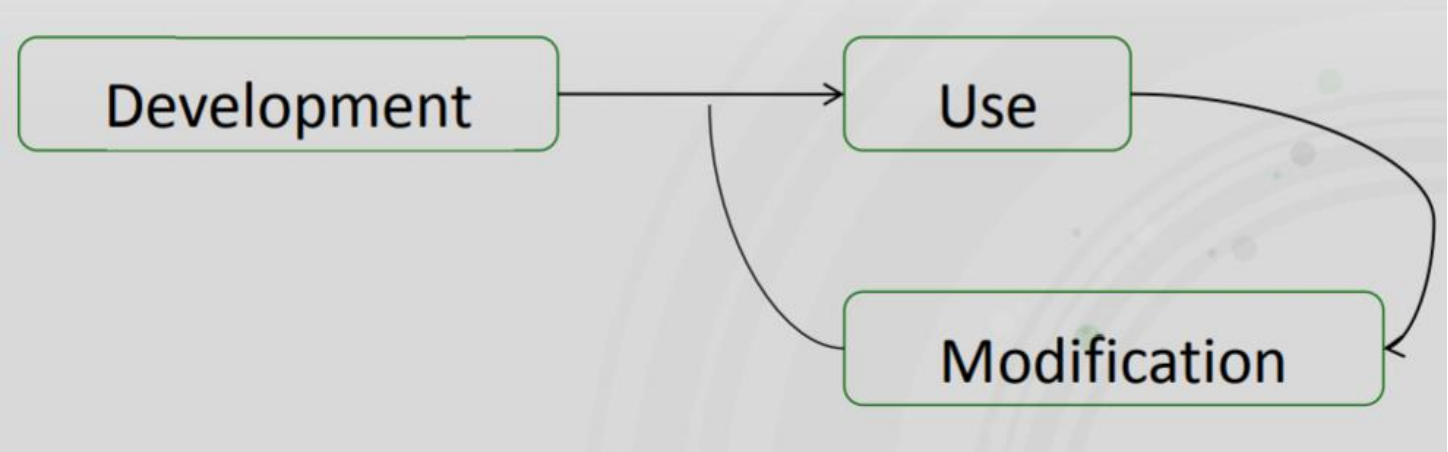
106

Summary

- Risk management life cycle:
  - Identify (risk identification)
  - Analyze (risk analysis)
  - Plan (contingency planning)
  - Track (risk monitoring)
  - Control (recovery management)

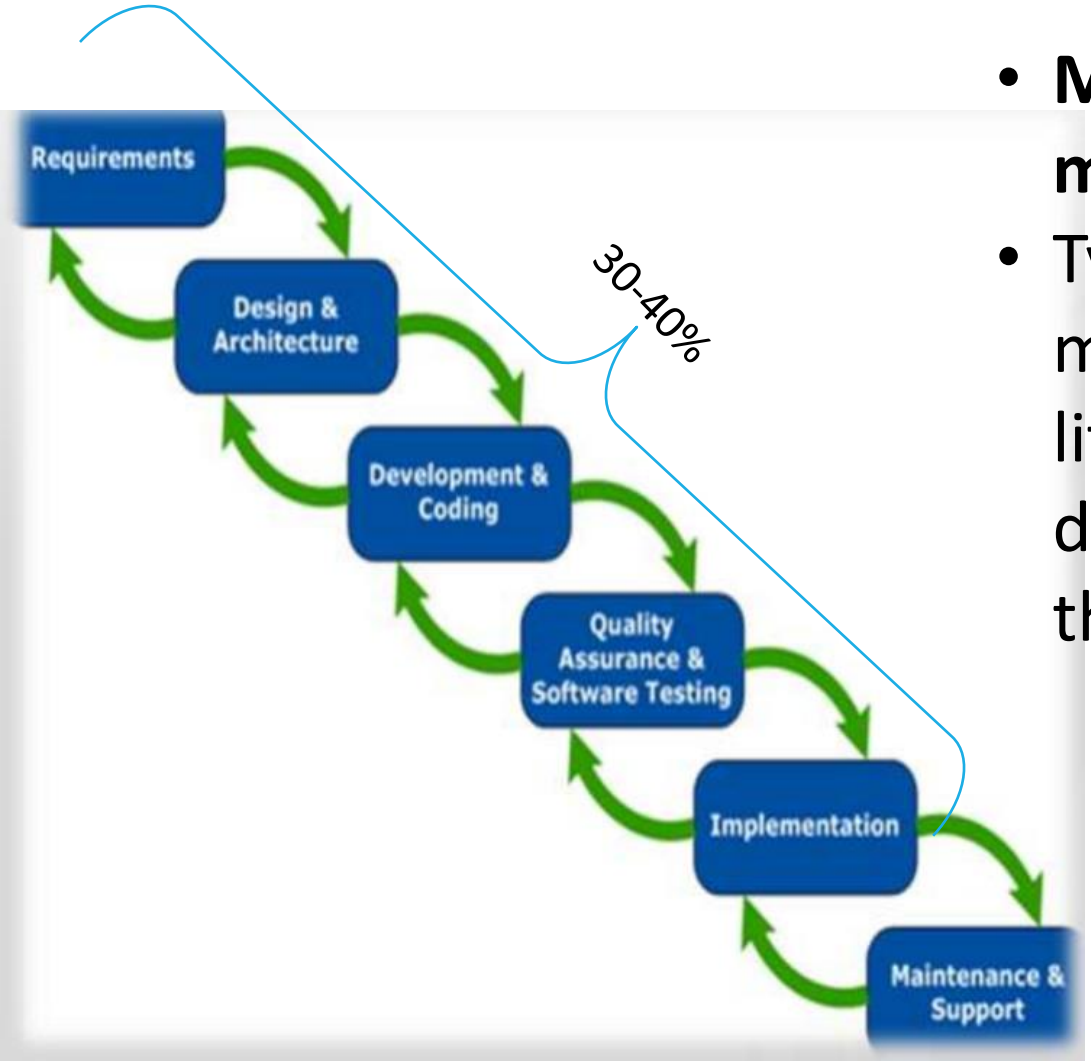
Software maintenance

- Once a Software is developed, its enters a cycle of being used and modified that continues for the rest of the Software's life
- Maintenance" includes all modifications made to a software system or product after it is delivered to the users



Software doesn't wear out.

Software maintenance



- **Maintenance cost twice as much as development.**
- Typically, software maintenance requires more lifecycle effort than initial development; perhaps in the ratio of 60:40 or 70:30

Software maintenance

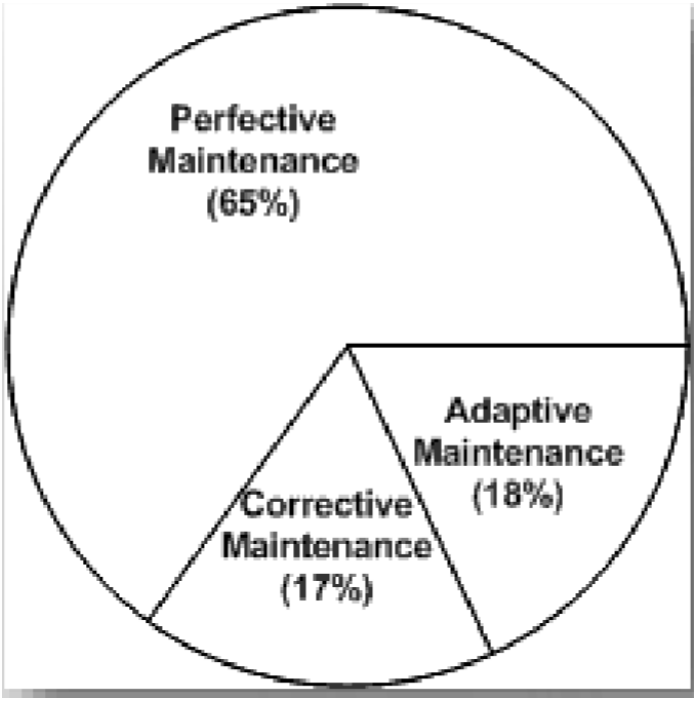
- Once we deploy our product, our job is still not finished.
- We and/or our customers are probably going to want to make some changes and improvements to the product.
- This is what we refer to as software maintenance.
- If we plan for maintainability all the way through the development cycle, we can make our job much easier when we are in the maintenance phase.
- If your product is used, users will ask for improvements.
- The more successful products have high maintenance budgets

Types of Maintenance

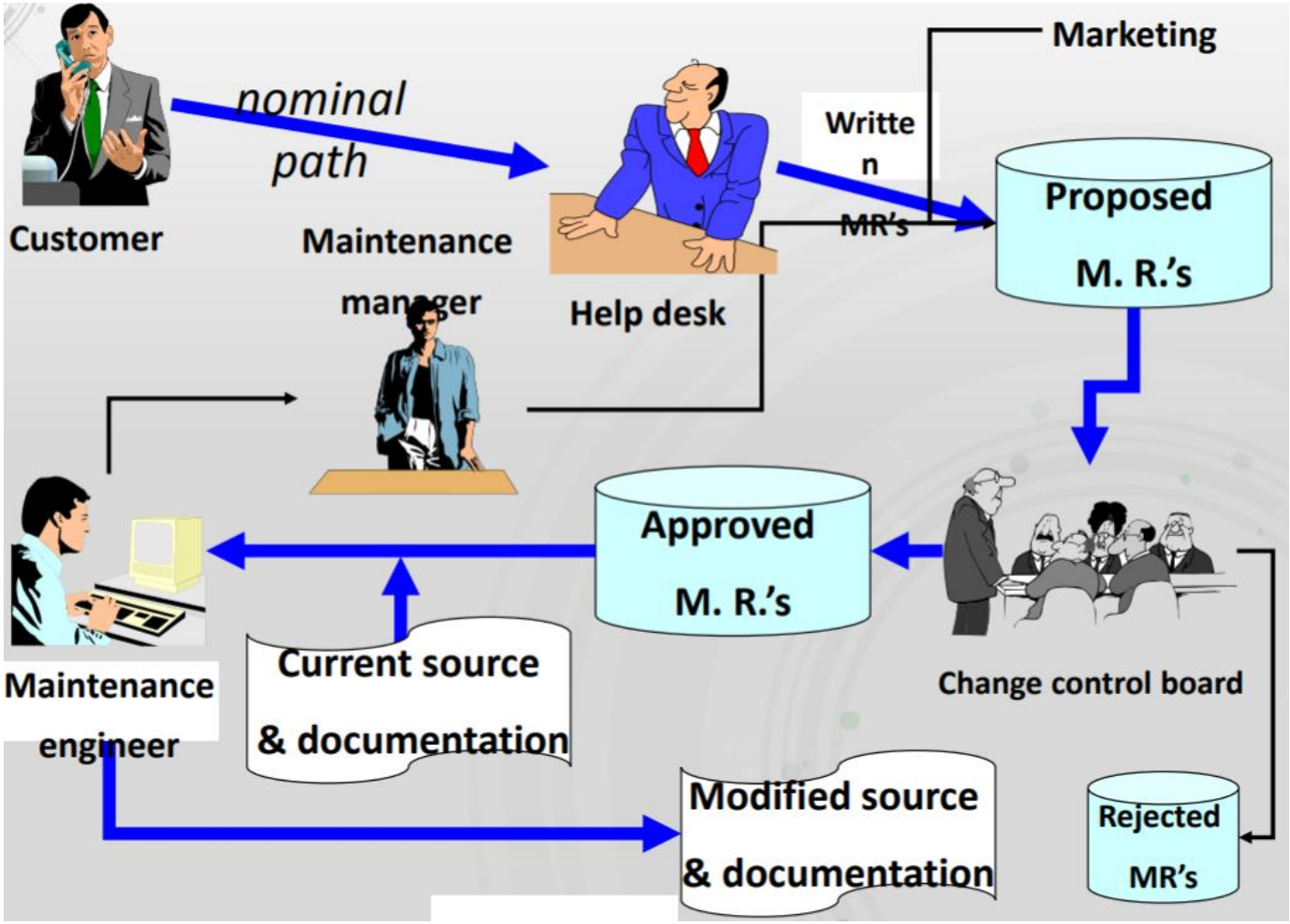
- Perfective maintenance – Adding or modifying the system's functionality to meet new requirements.
- Adaptive maintenance – Changing a system to adapt it to new hardware or operating system, such as database upgrades, operating system upgrades, compiler version changes etc.
- Corrective maintenance – Changing a system to fix coding, design, or requirements errors.

Maintenance Effort Required

- Maintenance Examples
- Operating System Patching
  - Microsoft, Apple, Linux/Unix
  - OS is core to use of computer, so it must be constantly maintained
- Commercial Software in General
  - customers need to be informed of updates
  - updates have to be easily available - web is good tool



A Typical Maintenance Flow



Software Supportability

- The capability of supporting a software system over its whole product life.
  - This implies satisfying any necessary needs or requirements, but also the provision of equipment, support infrastructure, additional software, facilities, manpower, or any other resource required to maintain the software and operational capacity of satisfying its function. ”
- Supportability is one of many quality factors that should be considered during the analysis and design actions

Software Supportability

- The software should contain facilities to assist support personnel when a defect is encountered in the operational environment (and make no mistake, defects will be encountered).
- Support personnel should have access to a database that contains records of all defects that have already been encountered — their characteristics, cause, and cure.“

Types of changes

Maintenance

- Repair software faults
  - Changing a system to correct deficiencies in the way meets its requirements.
- Adapt software to a different operating environment
  - Changing a system so that it operates in a different environment (computer, OS, etc.) from its initial implementation.
- Add to or modify the system's functionality
  - Modifying the system to satisfy new requirements.

reengineering

- Improve the program structure and system performance
  - Rewriting all or parts of the system to make it more efficient and maintainable.

Maintainability Factors

- Availability of qualified staff.
- Understandable system structure.
- Use of standardized programming languages and operating systems.
- Standardized structure of documentation.
- Availability of test cases.
- Built-in debugging facilities.
- Availability of a proper computer to conduct maintenance.

Maintenance Problems

- Developer not available.
- Proper documentation doesn't exist.
- Not designed for change.
- Maintenance activity not highly regarded

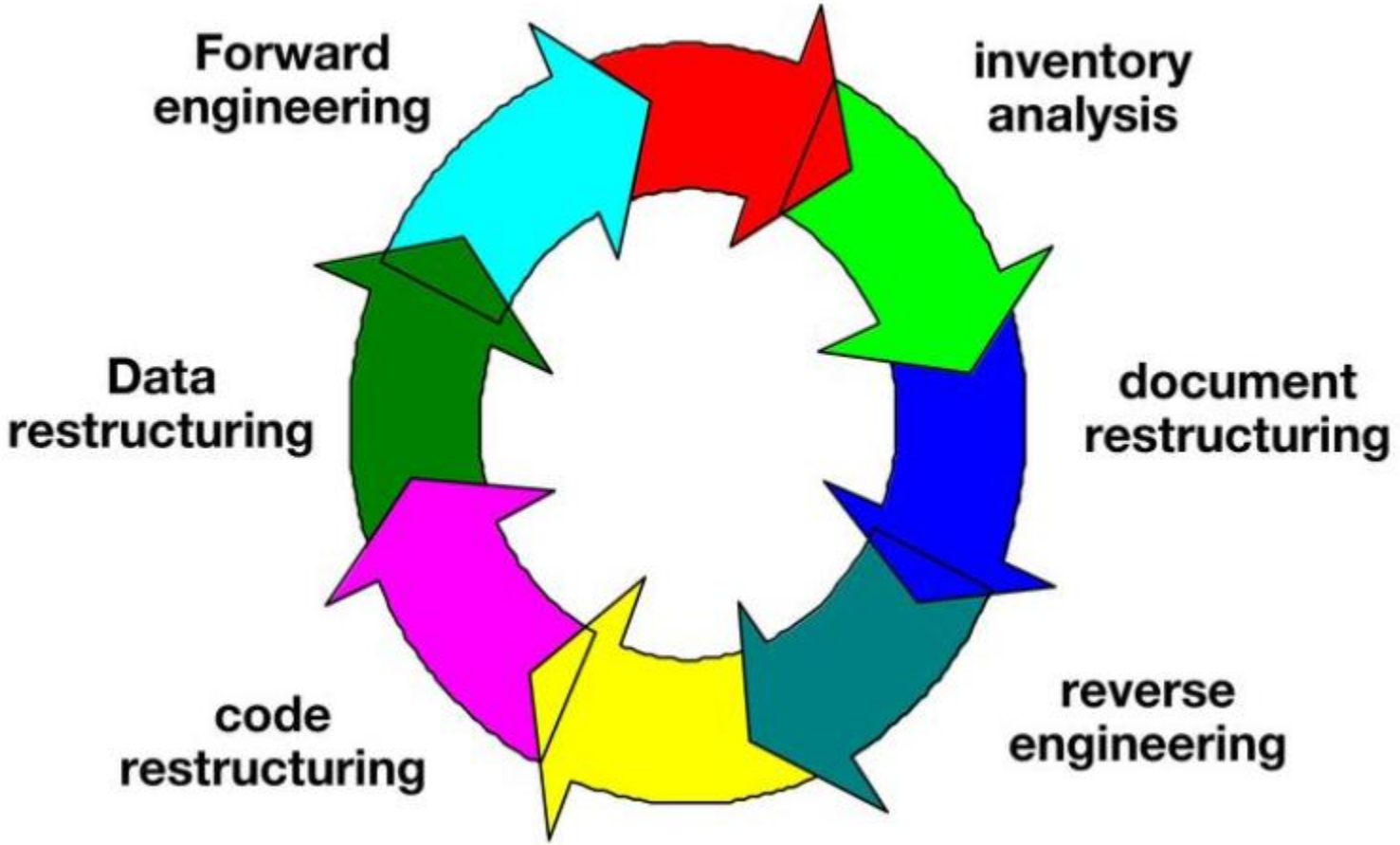
Maintenance Option

- Re-design, re-code and re-test those portions of the software that require modification.
- Completely re-design, re-code and re-test the entire program (again, using good software engineering).

Software reengineering

- Reengineering takes time, it costs significant amounts of money, and it absorbs resources that might be otherwise occupied on immediate concerns
- For all of these reasons, reengineering is not accomplished in a few months or even a few years.
- software reengineering process model defines six activities - a cyclical model.
  - each of the activities presented as a part of the paradigm may be revisited.
  - For any particular cycle, the process can terminate after any one of these activities.

Software reengineering process model



Inventory analysis

- build a table that contains all applications
- establish a list of criteria, eg,
  - name of the application
  - year it was originally created
  - number of substantive changes made to it
  - total effort applied to make these changes
  - date of last substantive change
  - Effort applied to make the last change
  - system (s) in which it resides
  - applications to which it interfaces, ...
- analyze and prioritize to select candidates for reengineering

Document Restructuring

- Weak documentation is the trademark of many legacy systems.
- But what do we do about it? What are our options?
- Options…
  - Creating documentation is far too time consuming.
    - If the system works, we'll live with what we have.
    - In some cases, this is the correct approach.
  - Documentation must be updated, but we have limited resources.
    - We'll use a “document when touched” approach.
    - It may not be necessary to fully redocument an application.
  - The system is business critical and must be fully redocumented.
    - Even in this case, an intelligent approach is to pare documentation to an essential minimum.

Reverse Engineering

- analyze program code to create a representation of the program at a higher level of abstraction (e.g. UML diagrams)
- design and specification recovery

Code Restructuring

- Source code is analyzed using a restructuring tool.
- Poorly design code segments are redesigned
- Violations of structured programming constructs are noted and code is then restructured (this can be done automatically)
- The resultant restructured code is reviewed and tested to ensure that no anomalies have been introduced
- Internal code documentation is updated.

Data Restructuring

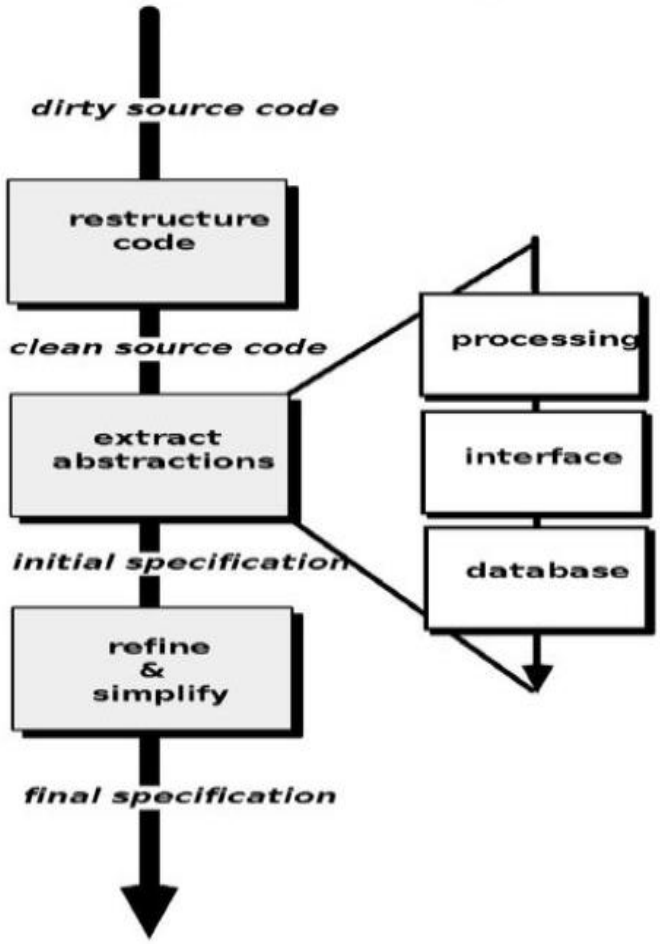
- Unlike code restructuring, which occurs at a relatively low level of abstraction, data structuring is a full-scale reengineering activity
- In most cases, data restructuring begins with a reverse engineering activity.
  - Current data architecture is dissected and necessary data models are defined.
  - Data objects and attributes are identified, and existing data structures are reviewed for quality.
  - When data structure is weak (eg, flat files are currently implemented, when a relational approach would greatly simplify processing), the data are reengineered
- Because data architecture has a strong influence on program architecture and the algorithms that populate it, changes to the data will invariably result in either architectural or code-level changes.

Forward Engineering

1. The cost to maintain a single line of source code may be 20 to 40 times the initial development cost for that line.
2. Re-design of software engineering (software and / or data structure), using modern design concepts, can greatly facilitate future maintenance.
3. Because a prototype of the software already exists, productivity and development should be well above average.
4. The user now has experience with the program. Therefore, new requirements and direction of change can be ascertained more easily.
5. Case tools for re-engineering automation of some parts of this post.
6. Configuration Complete programs (documentation, software and data) are present upon completion of preventive maintenance.

Reverse engineering

- Reverse engineering can extract design information from source code, considering the highly variable
  - the abstraction level,
  - the completeness of the documentation,
  - the degree to which tools and a human analyst work together,
  - the directionality of the process
- core of reverse engineering is an activity called extract abstractions



Reverse engineering - abstraction level

- The abstraction level should be as high as possible.
- the reverse engineering process should be capable of
  - deriving procedural design representations (a low-level abstraction),
  - program and data structure information (a somewhat higher level of abstraction),
  - object models,
  - data and/or control flow models (a relatively high level of abstraction),
  - data models (a high level of abstraction)
- As the abstraction level increases, you are provided with information that will allow easier understanding of the program

Reverse engineering - interactivity

- Completeness improves in direct proportion to the amount of analysis performed by the person doing reverse engineering.
- Interactivity refers to the degree to which the human is “integrated” with automated tools to create an effective reverse engineering process.
- As the abstraction level increases, interactivity must increase or completeness will suffer.

Reverse engineering - directionality

- If the directionality of the reverse engineering process is one-way, all information extracted from the source code is provided to the software engineer who can then use it during any maintenance activity.
- If directionality is two-way, the information is fed to a reengineering tool that attempts to restructure or regenerate the old program

Reverse engineering – document completeness

- Refers to the level of detail that is provided at an abstraction level
- Completeness decreases as the abstraction level increases
- For example, given a source code listing, it is relatively easy to develop a complete procedural design representation.
- Simple architectural design representations may also be derived, but it is far more difficult to develop a complete set of UML diagrams or models

Reverse Engineering to Understand Data

- Internal program data structures must often be reverse engineered at the program level
- Global data structures (e.g., files, databases) are often reengineered to accommodate new database management paradigms (e.g., the move from flat file to relational or object-oriented database systems) at the system level
- Database structure
- The following steps may be used to define the existing data model as a precursor to reengineering a new database model:
  - (1) build an initial object model,
  - (2) determine candidate keys (the attributes are examined to determine whether they are used to point to another record or table; those that serve as pointers become candidate keys),
  - (3) refine the tentative classes,
  - (4) define generalizations, and
  - (5) discover associations using techniques that are analogous to the CRC approach.

Reverse Engineering to Understand Processing

- To understand procedural abstractions, the code is analyzed at varying levels of abstraction: system, program, component, pattern, and statement

Reverse Engineering User Interfaces

- What are the basic actions (e.g., keystrokes and mouse clicks) that the interface must process?
- What is a compact description of the behavioral response of the system to these actions?
- What is meant by a “replacement,” or more precisely, what concept of equivalence of interfaces is relevant here?

RESTRUCTURING

- Code Restructuring
- Data Restructuring

Forward engineering

- You have the following options:

  1. You can struggle through modification after modification, fighting the ad hoc design and tangled source code to implement the necessary changes.

  2. You can attempt to understand the broader inner workings of the program in an effort to make modifications more effectively.

  3. You can redesign, recode, and test those portions of the software that require modification, applying a software engineering approach to all revised segments.

  4. You can completely redesign, recode, and test the program, using reengineering tools to assist us in understanding the current design.

Forward engineering

- Forward Engineering for Client-Server Architectures
  - Reengineering for client-server applications begins with a thorough analysis of the business environment that encompasses the existing mainframe.
  - Three layers of abstraction can be identified.
    - The database sits at the foundation of a client-server architecture and manages transactions and queries from server applications.
    - Yet these transactions and queries must be controlled within the context of a set of business rules (defined by an existing or reengineered business process).
    - Client applications provide targeted functionality to the user community.

Forward engineering

- Forward Engineering for Object-Oriented Architectures
  - the existing software is reverse engineered so that appropriate data, functional, and behavioral models can be created.
  - If the reengineered system extends the functionality or behavior of the original application, use cases are created.
  - The data models created during reverse engineering are then used in conjunction with CRC modeling to establish the basis for the definition of classes.
  - Class hierarchies, object-relationship models, object-behavior models, and subsystems are defined, and object-oriented design commences.