# 4. PROJECT PLANNING AND QUALITY MANAGEMENT

## PROJECT PLANNING

→ Project planning is one of the most important jobs of a software project manager.

→ The project plan, which is created at the start of a project, is used to communicate how the work will be done to the project team and customers, and to help assess progress on the project.

→ Project planning takes place at three stages in a project life cycle:

1. At the proposal stage, when there is bidding for a contract to develop or provide a software system, a plan may be required at this stage to help contractors decide if they have the resources to complete the work and to work out the price that they should quote to a customer.
2. During the project startup phase, there is a need to plan who will work on the project, how the project will be broken down into increments, how resources will be allocated across your company, etc.
3. Periodically throughout the project, when the plan is modified , in light of experience gained and information from monitoring the progress of the work, more information about the system being implemented and capabilities of development team are learnt.

→ This information allows to make more accurate estimates of how long the work will take.

## 4.1 Software Pricing

→ In principle, the price of a software product to a customer is simply the cost of development plus profit for the developer. Fig 4.1 shows the factors affecting software pricing

→ It is essential to think about organizational concerns, the risks associated with the project, and the type of contract that will be used.

→ These may cause the price to be adjusted upwards or downwards.

SVIT

→ Because of the organizational considerations involved, deciding on a project price should be a group activity involving marketing and sales staff, senior management, and project managers

| Factor | Description |
|---|---|
| Market opportunity | A development organization may quote a low price because it wishes to move into a new segment of the software market. Accepting a low profit on one project may give the organization the opportunity to make a greater profit later. The experience gained may also help it develop new products. |
| Cost estimate uncertainty | If an organization is unsure of its cost estimate, it may increase its price by a contingency over and above its normal profit. |
| Contractual terms | A customer may be willing to allow the developer to retain ownership of the source code and reuse it in other projects. The price charged may then be less than if the software source code is handed over to the customer. |
| Requirements volatility | If the requirements are likely to change, an organization may lower its price to win a contract. After the contract is awarded, high prices can be charged for changes to the requirements. |
| Financial health | Developers in financial difficulty may lower their price to gain a contract. It is better to make a smaller than normal profit or break even than to go out of business. Cash flow is more important than profit in difficult economic times. |

**Fig 4.1: Factors affecting software pricing**

# 4.2 Plan Driven Development

→ Plan-driven or plan-based development is an approach to software engineering where the development process is planned in detail.

→ A project plan is created that records the work to be done, who will do it, the development schedule, and the work products.

→ Managers use the plan to support project decision making and as a way of measuring progress.

→ Plan-driven development is based on engineering project management techniques and can be thought of as the 'traditional' way of managing large software development projects.

→ The principal argument against plan-driven development is that many early decisions have to be revised because of changes to the environment in which the software is to be developed and used

## 4.2.1 Project Plans

→ In a plan-driven development project, a project plan sets out the resources available to the project, the work breakdown, and a schedule for carrying out the work.

SVIT

→ The plan should identify risks to the project and the software under development, and the approach that is taken to risk management.

→ Plans normally include the following sections:

1. **Introduction:** Describes the objectives of the project and sets out the constraints (e.g., budget, time, etc.) that affect the management of the project.
2. **Project organization:** This describes the way in which the development team is organized, the people involved, and their roles in the team.
3. **Risk analysis:** This describes possible project risks, the likelihood of these risks arising, and the risk reduction strategies that are proposed.
4. **Hardware and software resource requirements:** This specifies the hardware and support software required to carry out the development. If hardware has to be bought, estimates of the prices and the delivery schedule may be included.
5. **Work breakdown:** This sets out the breakdown of the project into activities and identifies the milestones and deliverables associated with each activity.
6. **Project schedule:** Shows dependencies between activities, the estimated time required to reach each milestone, and the allocation of people to activities.
7. **Monitoring and reporting mechanisms:** This defines the management reports that should be produced, when these should be produced, and the project monitoring mechanisms to be used.

→ The project plan supplements are as shown in fig 4.2.

| Plan | Description |
|---|---|
| Quality plan | Describes the quality procedures and standards that will be used in a project. |
| Validation plan | Describes the approach, resources, and schedule used for system validation. |
| Configuration management plan | Describes the configuration management procedures and structures to be used. |
| Maintenance plan | Predicts the maintenance requirements, costs, and effort. |
| Staff development plan | Describes how the skills and experience of the project team members will be developed. |

**Fig 4.2: Project plan supplements**

## 4.2.2 The Planning Process

→ Project planning is an iterative process that starts when an initial project plan is created during the project startup phase.

→ Fig 4.3 is a UML activity diagram that shows a typical workflow for a project planning process.

→ At the beginning of a planning process, it is necessary to assess the constraints affecting the project.

→ These constraints are the required delivery date, staff available, overall budget, available tools, and so on.

→ It is also necessary to identify the project milestones and deliverables.

→ Milestones are points in the schedule against which progress can be accessed, for example, the handover of the system for testing.

→ Deliverables are work products that are delivered to the customer.

→ The process then enters a loop.

→ You draw up an estimated schedule for the project and the activities defined in the schedule are initiated or given permission to continue.

→ After some time (usually about two to three weeks), the progress must be reviewed and discrepancies must be noted from the planned schedule.

→ Because initial estimates of project parameters are inevitably approximate, minor slippages are normal and thereby modifications will have to be made to the original plan.

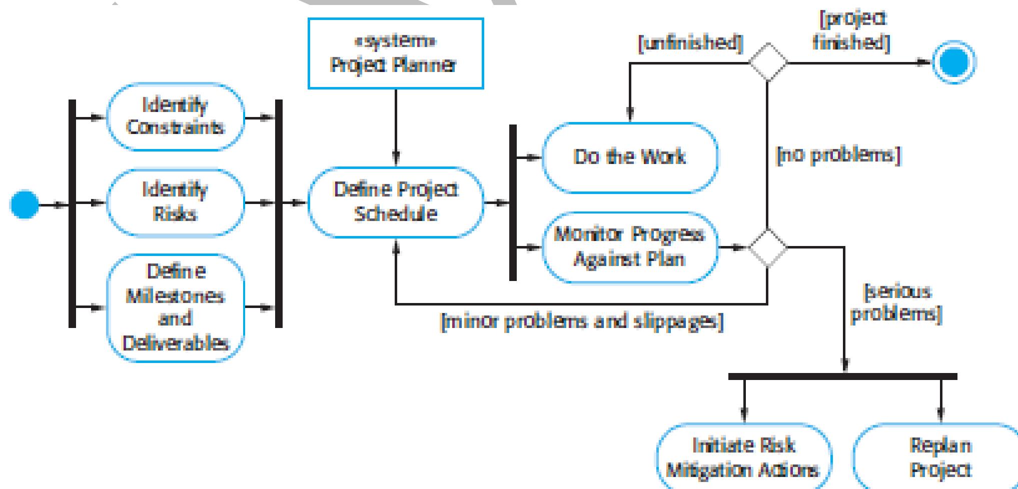


**Fig 4.3: The project planning process**

## 4.3 Project Scheduling

→ Project scheduling is the process of deciding how the work in a project will be organized as separate tasks, and when and how these tasks will be executed.

→ Here there is an estimation of the calendar time needed to complete each task, the effort required, and who will work on the tasks that have been identified.

→ It is essential to estimate the resources needed to complete each task, such as the disk space required on a server, the time required on specialized hardware, such as a simulator, and what the travel budget will be .

→ Scheduling in plan-driven projects (Fig 4.4) involves breaking down the total work involved in a project into separate tasks and estimating the time required to complete each task.

→ Tasks should normally last at least a week, and no longer than 2 months.

→ Finer subdivision means that a disproportionate amount of time must be spent on re-planning and updating the project plan.

→ The maximum amount of time for any task should be around 8 to 10 weeks.

→ If it takes longer than this, the task should be subdivided for project planning and scheduling.

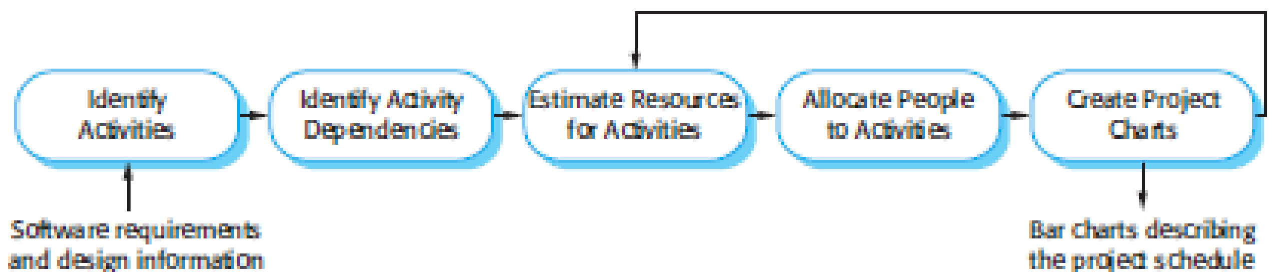


**Fig 4.4: The project scheduling process**

## 4.3.1 Schedule Representation

→ Project schedules may simply be represented in a table or spreadsheet showing the tasks, effort, expected duration, and task dependencies.

→ There are two types of representation that are commonly used:

1. Bar charts, which are calendar-based, show who is responsible for each activity, the expected elapsed time, and when the activity is scheduled to begin and end. Bar charts are sometimes called 'Gantt charts'.
2. Activity networks, which are network diagrams, show the dependencies between the different activities making up a project.

→ Project activities are the basic planning element. Each activity has:

1. A duration in calendar days or months.

2. An effort estimate, which reflects the number of person-days or person-months to complete the work.
3. A deadline by which the activity should be completed.
4. A defined endpoint. This represents the tangible result of completing the activity. This could be a document, the holding of a review meeting, the successful execution of all tests, etc.

→ When planning a project, milestones must also be defined; that is, each stage in the project where a progress assessment can be made.

→ Each milestone should be documented by a short report that summarizes the progress made and the work done.

→ Milestones may be associated with a single task or with groups of related activities.

→ For example, in fig 4.5, milestone M1 is associated with task T1 and milestone M3 is associated with a pair of tasks, T2 and T4.

→ A special kind of milestone is the production of a project deliverable.

→ A deliverable is a work product that is delivered to the customer. It is the outcome of a significant project phase such as specification or design.

→ Usually, the deliverables that are required are specified in the project contract and the customer's view of the project's progress depends on these deliverables.

| Task | Effort (person-days) | Duration (days) | Dependencies |
|---|---|---|---|
| T1 | 15 | 10 | |
| T2 | 8 | 15 | |
| T3 | 20 | 15 | T1 (M1) |
| T4 | 5 | 10 | |
| T5 | 5 | 10 | T2, T4 (M3) |
| T6 | 10 | 5 | T1, T2 (M4) |
| T7 | 25 | 20 | T1 (M1) |
| T8 | 75 | 25 | T4 (M2) |
| T9 | 10 | 15 | T3, T6 (M5) |
| T10 | 20 | 15 | T7, T8 (M6) |
| T11 | 10 | 10 | T9 (M7) |
| T12 | 20 | 10 | T10, T11 (M8) |

**Fig 4.5: Tasks, durations and dependencies**

→ Fig 4.6 takes the information in Figure 4.5 and presents the project schedule in a graphical format.

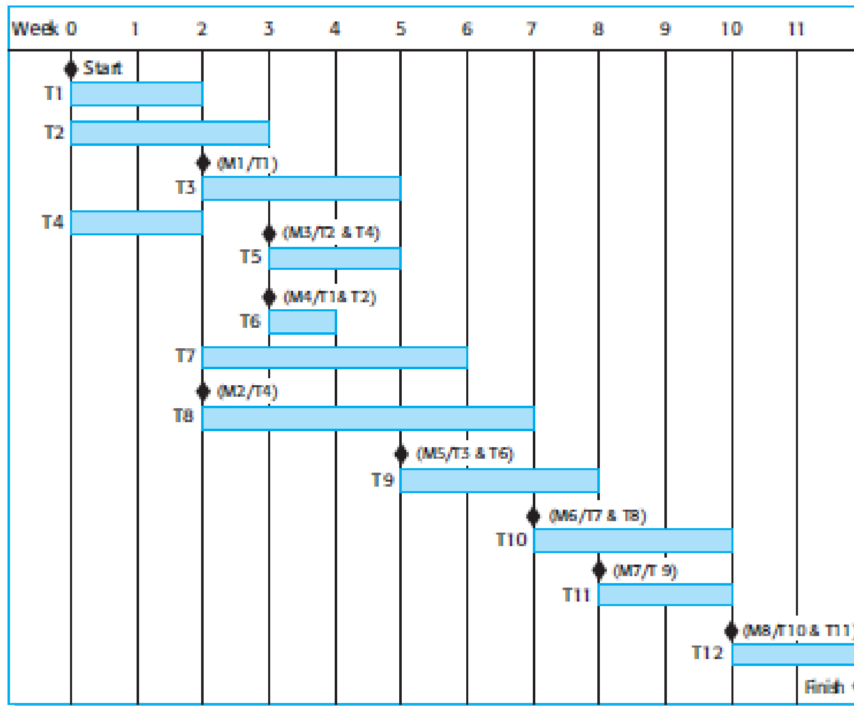


Fig 4.6: Activity bar chart

→ It is a bar chart showing a project calendar and the start and finish dates of tasks.

→ Reading from left to right, the bar chart clearly shows when tasks start and end.

→ The milestones (M1, M2, etc.) are also shown on the bar chart

→ In Fig 4.7, it is observed that Mary is a specialist, who works on only a single task in the project.

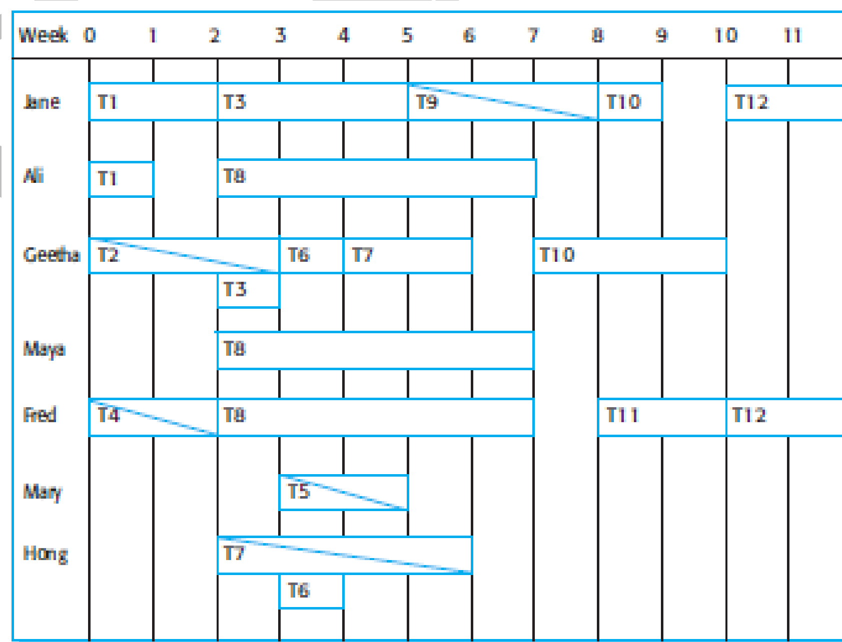


Fig 4.7: Staff allocation chart

→ This can cause scheduling problems. If one project is delayed while a specialist is working on it, this may have a knock-on effect on other projects where the specialist is also required.

→ These may then be delayed because the specialist is not available.

→ Delays can cause serious problems with staff allocation, especially when people are working on several projects at the same time.

→ If a task (T) is delayed, the people allocated may be assigned to other work (W).

→ To complete this may take longer than the delay but, once assigned, they cannot simply be reassigned back to the original task, T.

→ This may then lead to further delays in T as they complete W.

## 4.4 Estimation Techniques

→ Project schedule estimation is difficult.

→ There might be a need to make initial estimates on the basis of a high-level user requirements definition.

→ Organizations need to make software effort and cost estimates.

→ There are two types of technique that can be used to do this:

1. **Experience-based Techniques**: The estimate of future effort requirements is based on the manager's experience of past projects and the application domain.
2. **Algorithmic cost Modeling**: In this approach, a formulaic approach is used to compute the project effort based on estimates of product attributes, such as size, and process characteristics, such as experience of staff involved.

→ During development planning, estimates become more and more accurate as the project progresses (Fig 4.8).

→ Experience-based techniques rely on the manager's experience of past projects and the actual effort expended in these projects on activities that are related to software development.

→ The difficulty with experience-based techniques is that a new software project may not have much in common with previous projects

SVIT

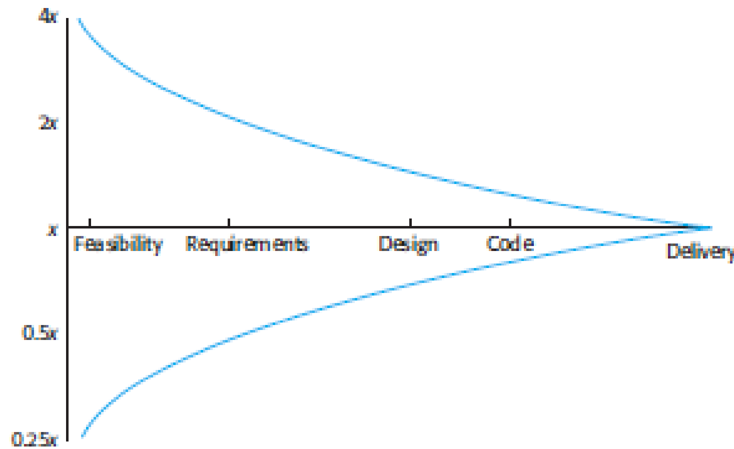


**Fig 4.8: Estimate Uncertainty**

## 4.4.1 Algorithmic Cost Modeling

→ Algorithmic cost modeling uses a mathematical formula to predict project costs based on estimates of the project size; the type of software being developed; and other team, process, and product factors.

→ An algorithmic cost model can be built by analyzing the costs and attributes of completed projects, and finding the closest-fit formula to actual experience.

→ Algorithmic models for estimating effort in a software project are mostly based on a simple formula:

$$\text{Effort} = A * \text{Size}^B * M$$

→ A is a constant factor which depends on local organizational practices and the type of software that is developed.

→ Size may be either an assessment of the code size of the software or a functionality estimate expressed in function or application points.

→ The value of exponent B usually lies between 1 and 1.5. M is a multiplier made by combining process, product, and development attributes, such as the dependability requirements for the software and the experience of the development team.

→ All algorithmic models have similar problems:

1. It is often difficult to estimate Size at an early stage in a project, when only the specification is available. Function-point and application-point estimates are easier to produce than estimates of code size but are still often inaccurate.
2. The estimates of the factors contributing to B and M are subjective. Estimates vary from one person to another, depending on their background and experience of the type of system that is being developed



SVIT

### 4.4.2 The COCOMO II Model

→ This is an empirical model that was derived by collecting data from a large number of software projects.

→ These data were analyzed to discover the formulae that were the best fit to the observations.

→ The COCOMO II model takes into account more modern approaches to software development, such as rapid development using dynamic languages, development by component composition, and use of database programming.

→ COCOMO II supports the spiral model of development.

→ The sub-models (Fig 4.9) that are part of the COCOMO II model are:

1. **An application-composition model**: Models the effort required to develop systems that are created from reusable components, scripting, or database programming. Software size estimates are based on application points, and a simple size/productivity formula is used to estimate the effort required.
2. **An early design model**: This model is used during early stages of the system design after the requirements have been established.
3. **A reuse model**: This model is used to compute the effort required to integrate reusable components and/or automatically generated program code. It is normally used in conjunction with the post-architecture model.
4. **A post-architecture model:** Once the system architecture has been designed, a more accurate estimate of the software size can be made. Again, this model uses the standard formula for cost estimation discussed above.

→ **The Application-Composition Model**

- The application-composition model was introduced into COCOMO II to support the estimation of effort required for prototyping projects and for projects where the software is developed by composing existing components.
- It is based on an estimate of weighted application points (sometimes called object points), divided by a standard estimate of application point productivity.
- The estimate is then adjusted according to the difficulty of developing each application point.
- Application composition usually involves significant software reuse.

SVIT

- It is almost certain that some of the application points in the system will be implemented using reusable components the final formula for effort computation for system prototypes is:

$$PM = (NAP * (1 - \%reuse / 100)) / PROD$$

Where,

PM → the effort estimate in person-months.

NAP → the total number of application points in the delivered system.

%reuse→an estimate of the amount of reused code in the development.

→ **The Early Design Model**

- This model may be used during the early stages of a project, before a detailed architectural design for the system is available.
- Early design estimates are most useful for option exploration where you need to compare different ways of implementing the user requirements.
- The estimates produced at this stage are based on the standard formula for algorithmic models, namely:

$$Effort = A * Size^{B} * M$$

- Boehm proposed that the coefficient A should be 2.94.
- The exponent B reflects the increased effort required as the size of the project increases.
- This can vary from 1.1 to 1.24 depending on the novelty of the project, the development flexibility, the risk resolution processes used, the cohesion of the development team, and the process maturity level of the organization.
- This results in an effort computation as follows:

$$PM = 2.94 * Size^{(1.1 - 1.24)} * M$$

Where

$$M = PERS * RCPX * RUSE * PDIF * PREX * FCIL * SCED$$

- The multiplier M is based on seven project and process attributes that increase or decrease the estimate.
- The attributes used in the early design model are product reliability and complexity (RCPX), reuse required (RUSE), platform difficulty (PDIF),

SVIT

personnel capability (PERS), personnel experience (PREX), schedule (SCED), and support facilities (FCIL).

→ **The Reuse Model**

- COCOMO II considers two types of reused code.
- 'Black-box' code is code that can be reused without understanding the code or making changes to it.
- The development effort for black-box code is taken to be zero.
- 'White box' code has to be adapted to integrate it with new code or other reused components.
- A model (often in UML) is analyzed and code is generated to implement the objects specified in the model.
- The COCOMO II reuse model includes a formula to estimate the effort required to integrate this generated code:

  $PM_{Auto}$ = (ASLOC * AT / 100) / ATPROD // Estimate for generated code

  Where,

  ASLOC → total number of lines of reused code, including code that is automatically generated.

  AT → percentage of reused code that is automatically generated.

  ATPROD → productivity of engineers in integrating such code.
- If there are a total of 20,000 lines of reused source code in a system and 30% of this is automatically generated, then the effort required to integrate the generated code is:

  (20.000 * 30/100) / 2400 = 2.5 person months //Generated Code
- The following formula is used to calculate the number of equivalent lines of source code:

  ESLOC = ASLOC * AAM

  Where,

  ESLOC → the equivalent number of lines of new source code.

  ASLOC → the number of lines of code in the components that have to be changed.

  AAM → an Adaptation Adjustment Multiplier (AAM) which adjusts the estimate to reflect the additional effort required to reuse code.

SVIT

* AAM is the sum of three components:
    i. An adaptation component (referred to as AAF) that represents the costs of making changes to the reused code. The adaptation component includes subcomponents that take into account design, code, and integration changes.
    ii. An understanding component (referred to as SU) that represents the costs of understanding the code to be reused and the familiarity of the engineer with the code. SU ranges from 50 for complex unstructured code to 10 for well-written, object-oriented code.
    iii. An assessment factor (referred to as AA) that represents the costs of reuse decision making. That is, some analysis is always required to decide whether or not code can be reused, and this is included in the cost as AA. AA varies from 0 to 8 depending on the amount of analysis effort required.

→ **The Post-Architecture Level**

* The post-architecture model is the most detailed of the COCOMO II models.
* It is used once an initial architectural design for the system is available so the subsystem structure is known.
* An estimation for each part of the system can then be made.
* The starting point for estimates produced at the post-architecture level is the same basic formula used in the early design estimates:
* Estimate of the code size can be done using three parameters:
    i. An estimate of the total number of lines of new code to be developed (SLOC).
    ii. An estimate of the reuse costs based on an equivalent number of source lines of code (ESLOC), calculated using the reuse model.
    iii. An estimate of the number of lines of code that are likely to be modified because of changes to the system requirements. The value of the exponent B is based on five factors, as shown in Fig 4.9 These factors are rated on a six-point scale from 0 to 5, where 0 means 'extra high' and 5 means 'very low'.

SVIT

| Scale factor | Explanation |
|---|---|
| Precedentedness | Reflects the previous experience of the organization with this type of project. Very low means no previous experience; extra-high means that the organization is completely familiar with this application domain. |
| Development flexibility | Reflects the degree of flexibility in the development process. Very low means a prescribed process is used; extra-high means that the client sets only general goals. |
| Architecture/risk resolution | Reflects the extent of risk analysis carried out. Very low means little analysis; extra-high means a complete and thorough risk analysis. |
| Team cohesion | Reflects how well the development team knows each other and work together. Very low means very difficult interactions; extra-high means an integrated and effective team with no communication problems. |
| Process maturity | Reflects the process maturity of the organization. The computation of this value depends on the CMM Maturity Questionnaire, but an estimate can be achieved by subtracting the CMM process maturity level from 5. |

**Fig 4.9: Scale factors used in the exponent computation in the post-architecture model**

* Possible values for the ratings used in exponent calculation are therefore:
    i. **Precedentedness**, rated low (4). This is a new project for the organization.
    ii. **Development flexibility**, rated very high (1). No client involvement in the development process so there are few externally imposed changes
    iii. **Architecture/risk resolution**, rated very low (5). There has been no risk analysis carried out.
    iv. **Team cohesion**, rated nominal (3). This is a new team so there is no information available on cohesion.
    v. **Process maturity**, rated nominal (3). Some process control is in place.
* Fig 4.10 shows how the cost driver attributes can influence effort estimates.

| | |
|---|---|
| Exponent value | 1.17 |
| System size (including factors for reuse and requirements volatility) | 128,000 DSI |
| **Initial COCOMO estimate without cost drivers** | **730 person-months** |
| Reliability | Very high, multiplier = 1.39 |
| Complexity | Very high, multiplier = 1.3 |
| Memory constraint | High, multiplier = 1.21 |
| Tool use | Low, multiplier = 1.12 |
| Schedule | Accelerated, multiplier = 1.29 |
| **Adjusted COCOMO estimate** | **2,306 person-months** |
| Reliability | Very low, multiplier = 0.75 |
| Complexity | Very low, multiplier = 0.75 |
| Memory constraint | None, multiplier = 1 |
| Tool use | Very high, multiplier = 0.72 |
| Schedule | Normal, multiplier = 1 |
| **Adjusted COCOMO estimate** | **295 person-months** |

**Fig 4.10: The effect of cost drivers on effort estimates**

### 4.4.3 Project Duration and Staffing

→ The COCOMO model includes a formula to estimate the calendar time required to complete a project:

→ TDEV is the nominal schedule for the project, in calendar months, ignoring any multiplier that is related to the project schedule.

→ PM is the effort computed by the COCOMO model. B is the complexity-related Exponent

→ There is a complex relationship between the number of people working on a project, the effort that will be devoted to the project, and the project delivery schedule.

→ If four people can complete a project in 13 months (i.e., 52 person-months of effort), then you might think that by adding one more person, you can complete the work in 11 months (55 person-months of effort).

→ The COCOMO model suggests that you will, in fact, need six people to finish the work in 11 months (66 person-months of effort).

→ The reason for this is that adding people actually reduces the productivity of existing team members and so the actual increment of effort added is less than one person.

→ As the project team increases in size, team members spend more time communicating and defining interfaces between the parts of the system developed by other people.

→ Doubling the number of staff (for example) therefore does not mean that the duration of the project will be halved

SVIT

# QUALITY MANAGEMENT

→ Software quality management for software systems has three principal concerns:

1. At the organizational level, quality management is concerned with establishing a framework of organizational processes and standards that will lead to high quality software. This means that the quality management team should take responsibility for defining the software development processes to be used and standards that should apply to the software and related documentation, including the system requirements, design, and code.
2. At the project level, quality management involves the application of specific quality processes, checking that these planned processes have been followed,

and ensuring that the project outputs are conformant with the standards that are applicable to that project.

3. Quality management at the project level is also concerned with establishing a quality plan for a project. The quality plan should set out the quality goals for the project and define what processes and standards are to be used.

→ Quality assurance (QA) is the definition of processes and standards that should lead to high-quality products and the introduction of quality processes into the manufacturing process.

→ Quality control is the application of these quality processes to weed out products that are not of the required level of quality.

→ Quality management provides an independent check on the software development process.

→ The quality management process checks the project deliverables to ensure that they are consistent with organizational standards and goals

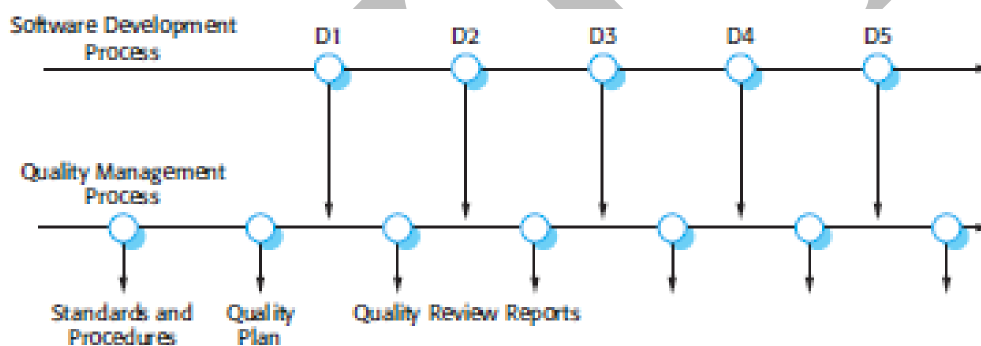


**Fig 4.11: Quality management and software development**

→ An outline structure for quality plan includes:

1. **Product introduction:** A description of the product, its intended market, and the quality expectations for the product.
2. **Product plans:** The critical release dates and responsibilities for the product, along with plans for distribution and product servicing
3. **Process descriptions:** The development and service processes and standards that should be used for product development and management.
4. **Quality goals:** The quality goals and plans for the product, including an identification and justification of critical product quality attributes.
5. **Risks and risk management:** The key risks that might affect product quality and the actions to be taken to address these risks

# 4.5 Software Quality

→ Different software quality attributes are as shown below

| Safety | Understandability | Portability |
|---|---|---|
| Security | Testability | Usability |
| Reliability | Adaptability | Reusability |
| Resilience | Modularity | Efficiency |
| Robustness | Complexity | Learnability |

**Fig 4.12: Software quality attributes**

→ A manufacturing process involves configuring, setting up, and operating the machines involved in the process.

→ Once the machines are operating correctly, product quality naturally follows

→ The quality of the product is measured and the process is changed until the quality level needed is achieved.

→ Fig 4.13 illustrates this process-based approach to achieving product quality.

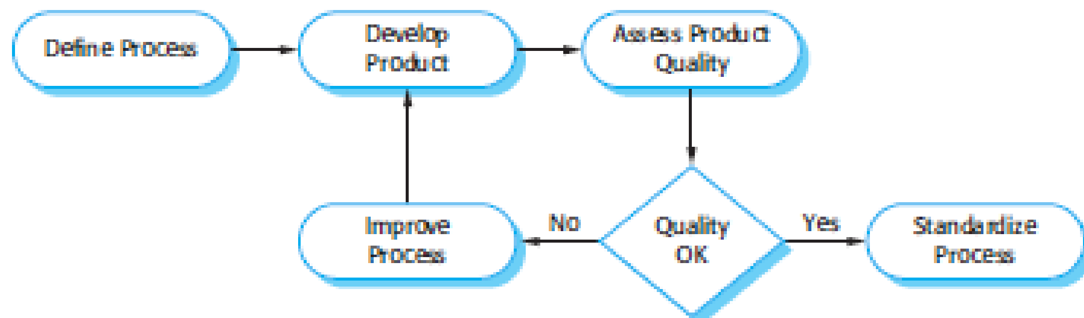


**Fig 4.13: Process based quality**

# 4.6 Software Standards

→ Software standards are important for three reasons:

1. Standards capture wisdom that is of value to the organization. They are based on knowledge about the best or most appropriate practice for the company. This knowledge is often only acquired after a great deal of trial and error.
2. Standards provide a framework for defining what 'quality' means in a particular setting. This depends on setting standards that reflect user expectations for software dependability, usability, and performance.

3. Standards assist continuity when work carried out by one person is taken up and continued by another. Standards ensure that all engineers within an organization adopt the same practices.

→ There are two related types of software engineering standard that may be defined and used in software quality management:

1. **Product Standards:**
    * These apply to the software product being developed.
    * They include document standards, such as the structure of requirements documents, documentation standards, such as a standard comment header for an object class definition, and coding standards, which define how a programming language should be used.
2. **Process Standards:**
    * These define the processes that should be followed during software development.
    * They should encapsulate good development practice.
    * Process standards may include definitions of specification, design and validation processes, process support tools, and a description of the documents that should be written during these processes.

## 4.7 Reviews and Inspections

→ Reviews and inspections are QA activities that check the quality of project deliverables.

→ This involves examining the software, its documentation and records of the process to discover errors and omissions and to see if quality standards have been followed.

→ During a review, a group of people examine the software and its associated documentation, looking for potential problems and non-conformance with standards.

→ The review team makes informed judgments about the level of quality of a system or project deliverable.

→ Project managers may then use these assessments to make planning decisions and allocate resources to the development process.

→ Quality reviews are based on documents that have been produced during the software development process.

SVIT

→ The purpose of reviews and inspections is to improve software quality, not to assess the performance of people in the development team.

→ Reviewing is a public process of error detection, compared with the more private component-testing process

### 4.7.1 The Review Process

→ Review process is structured into 3 phases:

1. **Pre-Review Activities:**
    * These are preparatory activities that are essential for the review to be effective.
    * Pre-review activities are concerned with review planning and review preparation.
    * Review planning involves setting up a review team, arranging a time and place for the review, and distributing the documents to be reviewed.
    * During review preparation, the team may meet to get an overview of the software to be reviewed
2. **The Review Meeting**:
    * During the review meeting, an author of the document or program being reviewed should 'walk through' the document with the review team.
    * The review itself should be relatively short—two hours at most. One team member should chair the review and another should formally record all review decisions and actions to be taken.
3. **Post-Review Activities**:
    * After a review meeting has finished, the issues and problems raised during the review must be addressed.
    * This may involve fixing software bugs, refactoring software so that it conforms to quality standards, or rewriting documents.

### 4.7.2 Program Inspections

→ Program inspections are 'peer reviews' where team members collaborate to find bugs in the program that is being developed.

→ Inspections may be part of the software verification and validation processes.

SVIT

→ They complement testing as they do not require the program to be executed.

→ This means that incomplete versions of the system can be verified and that representations such as UML models can be checked.

→ During an inspection, a checklist of common programming errors is often used to focus the search for bugs.

→ This checklist may be based on examples from books or from knowledge of defects that are common in a particular application domain.

→ Possible checks are as shown in fig 4.14.

| Fault class | Inspection check |
|---|---|
| Data faults | • Are all program variables initialized before their values are used?<br>• Have all constants been named?<br>• Should the upper bound of arrays be equal to the size of the array or Size -1?<br>• If character strings are used, is a delimiter explicitly assigned?<br>• Is there any possibility of buffer overflow? |
| Control faults | • For each conditional statement, is the condition correct?<br>• Is each loop certain to terminate?<br>• Are compound statements correctly bracketed?<br>• In case statements, are all possible cases accounted for?<br>• If a break is required after each case in case statements, has it been included? |
| Input/output faults | • Are all input variables used?<br>• Are all output variables assigned a value before they are output?<br>• Can unexpected inputs cause corruption? |
| Interface faults | • Do all function and method calls have the correct number of parameters?<br>• Do formal and actual parameter types match?<br>• Are the parameters in the right order?<br>• If components access shared memory, do they have the same model of the shared memory structure? |
| Storage management faults | • If a linked structure is modified, have all links been correctly reassigned?<br>• If dynamic storage is used, has space been allocated correctly?<br>• Is space explicitly deallocated after it is no longer required? |
| Exception management faults | • Have all possible error conditions been taken into account? |

**Fig 4.14: An inspection checklist**

# 4.8 Software Measurement and Metrics

→ Software measurement is concerned with deriving a numeric value or profile for an attribute of a software component, system, or process.

→ The long-term goal of software measurement is to use measurement in place of reviews to make judgments about software quality.

→ Using software measurement, a system could ideally be assessed using a range of metrics and, from these measurements, a value for the quality of the system could be inferred.

→ Software metric is a characteristic of a software system, system documentation, or development process that can be objectively measured.

→ Examples of metrics include the size of a product in lines of code.

→ Software metrics may be either control metrics or predictor metrics.

→ Control metrics support process management, and predictor metrics helps to predict characteristics of the software.

→ Control metrics are usually associated with software processes.

→ Examples of control or process metrics are the average effort and the time required to repair reported defects.

→ Predictor metrics are associated with the software itself and are sometimes known as 'product metrics.

→ Both control and predictor metrics may influence management decision making, as shown in fig 4.15.

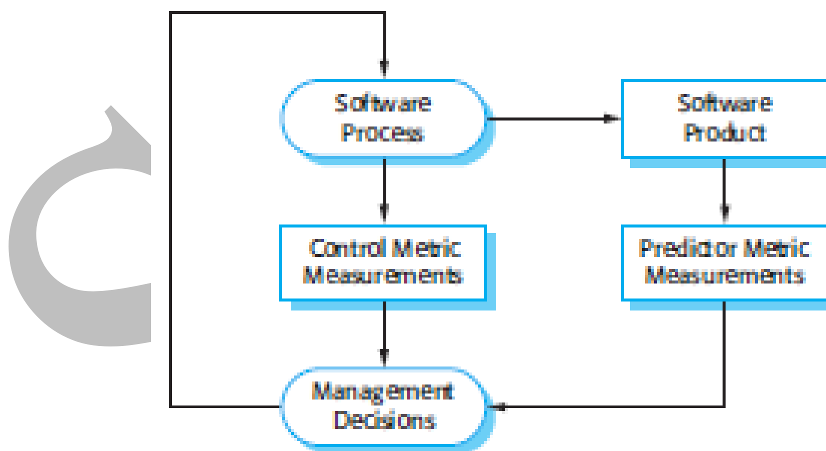


**Fig 4.15: Predictor and control measurements**

→ There are two ways in which measurements of a software system may be used:

1. **To assign a value to system quality attributes***:* By measuring the characteristics of system components, such as their cyclomatic complexity, and then aggregating these measurements, you can assess system quality attributes, such as maintainability.
2. **To identify the system components whose quality is substandard:** Measurements can identify individual components with characteristics that

deviate from the norm. For example, components can be measured to discover those with the highest complexity.

→ Fig 4.16 shows some external software quality attributes and internal attributes that could, intuitively, be related to them.

→ The diagram suggests that there may be relationships between external and internal attributes, but it does not say how these attributes are related.

→ If the measure of the internal attribute is to be a useful predictor of the external software characteristic, three conditions must hold

1. The internal attribute must be measured accurately. This is not always straightforward and it may require special-purpose tools to make the measurements.
2. A relationship must exist between the attribute that can be measured and the external quality attribute that is of interest. That is, the value of the quality attribute must be related, in some way, to the value of the attribute than can be measured.
3. This relationship between the internal and external attributes must be understood, validated, and expressed in terms of a formula or model. Model formulation involves identifying the functional form of the model (linear, exponential, etc.)by analysis of collected data, identifying the parameters that are to be included in the model, and calibrating these parameters using existing data.

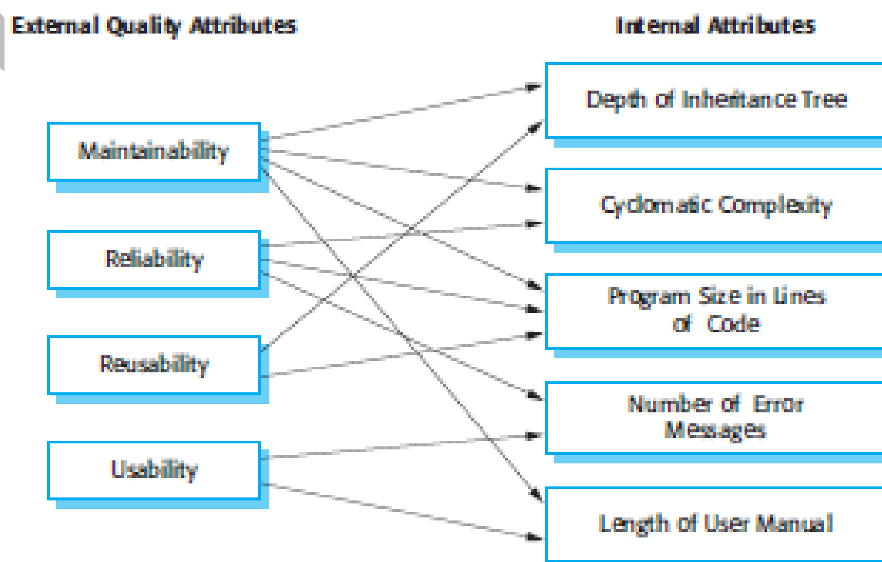


**Fig 4.16: Relationships between internal and external software**

### 4.8.1 Product Metrics

→ Product metrics are predictor metrics that are used to measure internal attributes of a software system.

→ Examples of product metrics include the system size, measured in lines of code, or the number of methods associated with each object class.

→ Product metrics fall into two classes:

1. Dynamic metrics, which are collected by measurements made of a program in execution. These metrics can be collected during system testing or after the system has gone into use. An example might be the number of bug reports or the time taken to complete a computation.
2. Static metrics, which are collected by measurements made of representations of the system, such as the design, program, or documentation. Examples of static metrics are the code size and the average length of identifiers used.

→ The metrics in fig 4.17 are applicable to any program but more specific object-oriented (OO) metrics have also been proposed

| Software metric | Description |
|---|---|
| Fan-in/Fan-out | Fan-in is a measure of the number of functions or methods that call another function or method (say X). Fan-out is the number of functions that are called by function X. A high value for fan-in means that X is tightly coupled to the rest of the design and changes to X will have extensive knock-on effects. A high value for fan-out suggests that the overall complexity of X may be high because of the complexity of the control logic needed to coordinate the called components. |
| Length of code | This is a measure of the size of a program. Generally, the larger the size of the code of a component, the more complex and error-prone that component is likely to be. Length of code has been shown to be one of the most reliable metrics for predicting error-proneness in components. |
| Cyclomatic complexity | This is a measure of the control complexity of a program. This control complexity may be related to program understandability. I discuss cyclomatic complexity in Chapter 8. |
| Length of identifiers | This is a measure of the average length of identifiers (names for variables, classes, methods, etc.) in a program. The longer the identifiers, the more likely they are to be meaningful and hence the more understandable the program. |
| Depth of conditional nesting | This is a measure of the depth of nesting of if-statements in a program. Deeply nested if-statements are hard to understand and potentially error-prone. |
| Fog index | This is a measure of the average length of words and sentences in documents. The higher the value of a document's Fog index, the more difficult the document is to understand. |

**Fig 4.17: Static software product metrics**

→ Fig 4.18 summarizes Chidamber and Kemerer's suite

| Object-oriented metric | Description |
|---|---|
| Weighted methods per class (WMC) | This is the number of methods in each class, weighted by the complexity of each method. Therefore, a simple method may have a complexity of 1, and a large and complex method a much higher value. The larger the value for this metric, the more complex the object class. Complex objects are more likely to be difficult to understand. They may not be logically cohesive, so cannot be reused effectively as superclasses in an inheritance tree. |
| Depth of inheritance tree (DIT) | This represents the number of discrete levels in the inheritance tree where subclasses inherit attributes and operations (methods) from superclasses. The deeper the inheritance tree, the more complex the design. Many object classes may have to be understood to understand the object classes at the leaves of the tree. |
| Number of children (NOC) | This is a measure of the number of immediate subclasses in a class. It measures the breadth of a class hierarchy, whereas DIT measures its depth. A high value for NOC may indicate greater reuse. It may mean that more effort should be made in validating base classes because of the number of subclasses that depend on them. |
| Coupling between object classes (CBO) | Classes are coupled when methods in one class use methods or instance variables defined in a different class. CBO is a measure of how much coupling exists. A high value for CBO means that classes are highly dependent, and therefore it is more likely that changing one class will affect other classes in the program. |
| Response for a class (RFC) | RFC is a measure of the number of methods that could potentially be executed in response to a message received by an object of that class. Again, RFC is related to complexity. The higher the value for RFC, the more complex a class and hence the more likely it is that it will include errors. |
| Lack of cohesion in methods (LCOM) | LCOM is calculated by considering pairs of methods in a class. LCOM is the difference between the number of method pairs without shared attributes and the number of method pairs with shared attributes. The value of this metric has been widely debated and it exists in several variations. It is not clear if it really adds any additional, useful information over and above that provided by other metrics. |

**Fig 4.18: The CK object oriented metrics suite**

## 4.8.2 Software Component Analysis

→ A measurement process that may be part of a software quality assessment process is shown in fig 4.19.

→ Each system component can be analyzed separately using a range of metrics.

→ The values of these metrics may then be compared for different components and, perhaps, with historical measurement data collected on previous projects.

→ The key stages in this component measurement process are:

1. **Choose measurements to be made**:
   * The questions that the measurement is intended to answer should be formulated and the measurements required to answer these questions defined.

- Measurements that are not directly relevant to these questions need not be collected.

2. **Select components to be assessed**:
   - You may not need to assess metric values for all of the components in a software system.
   - Sometimes, a representative selection of components can be selected for measurement, allowing to make an overall assessment of system quality.
3. **Measure component characteristics:**
   - The selected components are measured and the associated metric values computed.
   - This normally involves processing the component representation (design, code, etc.) using an automated data collection tool.
   - This tool may be specially written or may be a feature of design tools that are already in use.
4. **Identify anomalous measurements**:
   - After the component measurements have been made, it can be compared with each other and to previous measurements that have been recorded in a measurement database.
5. **Analyze anomalous components**:
   - When the components that have anomalous values for chosen metrics have been identified, it becomes necessary to examine them to decide whether or not these anomalous metric values mean that the quality of the component is compromised.
   - An anomalous metric value for complexity

SVIT

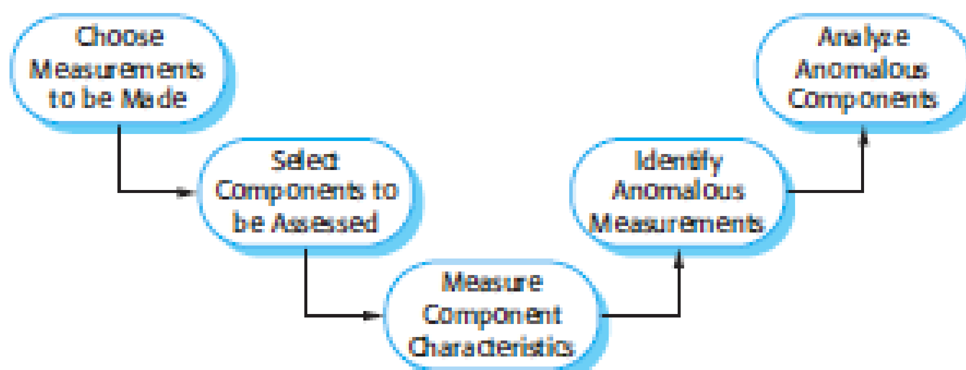


**Fig 4.19: The process of product measurement**

## 4.8.3 Measurement Ambiguity

→ When you collect quantitative data about software and software processes, the data must be analyzed in order to be understood.

→ It is easy to misinterpret data and to make inferences that are incorrect.

→ There are several reasons for the users to make change requests:

1. The software is not good enough and does not do what customers want it to do. They therefore request changes to deliver the functionality that they require.
2. Alternatively, the software may be very good and so it is widely and heavily used. Change requests may be generated because there are many software users who creatively think of new things that could be done with the software.

SVIT