Unit 2: Software Project Management

Introduction

Building computer software is a complex undertaking task, which particularly involves many people working over a relatively long time. That's why software projects needs to be managed. The software project management is the first layer of software engineering process. It starts before the technical work starts, continues as the software evolves from conceptual stage to implementation stage. It is a crucial activity because the success and failure of the software is directly depends on it. Software project management is needed because professional software engineering is always subject to budget constraints, schedule constraints and quality oriented focus.

Definition

Project management involves the planning, monitoring and control of the people, process and events that occurs as software evolves from a preliminary concept to an operational implementation. Software project management is an umbrella activity within software engineering. It begins before any technical activity is initiated and continues throughout the definition, development and support of computer software. The project management activity encompasses measurements and metrics estimation, risk analysis, schedules, tracking and control.

Effective software project management focuses on the four P's: People, Product, Process, and Project. The order is not arbitrary. The manager who forgets that software engineering work is an intensely human endeavor will never have success in project management. A manager who fails to encourage comprehensive customer communication early in the evolution of a project risks building an elegant solution for the wrong problem. The manager who pays little attention to the process runs the risk of inserting competent technical methods and tools into a vacuum. The manager who embarks without a solid project plan jeopardizes the success of the product.

1. **The People**
   - The "people factor" is so important that the Software Engineering Institute has developed a people management capability maturity model (PM-CMM).
   - To enhance the readiness of software organizations to undertake increasingly complex applications by helping to attract, grow, motivate, deploy, and retain the talent needed to improve their software development capability.
   - The **PM-CMM** is a companion to the software capability maturity model that guides organizations in the creation of a mature software process.
   - The PMCMM defines the following areas for software people.
     - Recruiting
     - Selection
     - Performance Management
     - Training
     - Career Development
     - Team Culture Development

2. **The Product**

Before a project can be planned, the Product objectives and scope should be established.

- Alternative solutions should be considered.
- Technical and management constraints should be identified.
- The software developer and customer must meet to define product objectives and scope.
- Objectives identify the overall goals for the product (from the customer's point of view) without considering how these goals will be achieved.
- Scope identifies the primary data, functions and behaviors that characterize the product, and more important, attempts to bound these characteristics in a quantitative manner.

3. **The Process**
   - A software process provides the framework from which a comprehensive plan for software development can be established.
   - A small number of framework activities are applicable to all software projects, regardless of their size or complexity.
   - A number of different tasks set—tasks, milestones, work products and quality assurance points—enable the framework activities to be adapted to the characteristics of the software project and the requirements of the project team.
   - Umbrella activities—such as software quality assurance, software configuration management, and measurement—overlay the process model.

4. **The Project**
   - We conduct planned and controlled software projects for one primary reason—it is the only known way to manage complexity.
   - The overall development cycle is called as Project.
   - In order to avoid project failure, a software project manager and the software engineers who build the product must avoid a set of common warning signs, understand the critical success factors that lead to good project management, and develop a commonsense approach for planning, monitoring and controlling the project.

The Software Team

The best team structure for any particular project depends on the nature of the project & product and also the individual characteristics of the team members. The basic team structures are as follows – a) Democratic teams, b) Chief programmer teams, c) Hierarchical team.

1. **Democratic Teams or Democratic decentralized (DD)**

It has following features:

- The team leader position does not rotate among the team members because a team functions best when one individual is responsible for coordinating team activities and for making final decisions in situations where collective decisions cannot work.

- Here all the decisions are made by collective effort of the members.
- All the activities carried out during project are collectively discussed and handled.

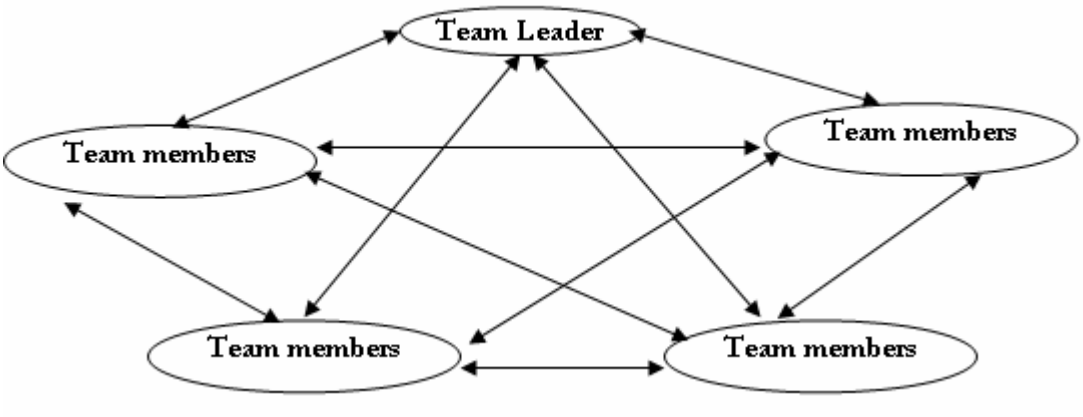

Figure: Democratic Team Structure

Advantages

- Opportunity for team members to contribute to decisions.
- Opportunity for team members to learn from each other.
- Increased job satisfaction due to equal importance and non-threatening environment.
- These teams can stay together for several years and may work on several different projects.

Disadvantages

- Communications overhead required for reaching to collective decisions.
- A lot of coordination required between team members.
- Less individual responsibility and authority results in less personal drive and initiative from team members.

2. Chief Programmer Teams or Controlled decentralized (CD)

It has following features:

- They are highly structured
- The chief programmer designs the product and makes all the decisions.
- The chief programmer implements the critical parts of the project.
- The chief programmer allocates the work for the individual programmer under him.
- Usually the number of programmers ranges from 2 to 5 only.
- The programmers do the coding, debug; document and unit test the system.
- The chief programmer is assisted by a backup consultant programmer on various technical problems, provides connection with the customer provides interaction with quality assurance group and may participate in analysis, design and implementation phases.

- The chief programmer is also assisted by an administrative program manager, who handles the administrative details which includes time cards for the employees, sick leave and vacation schedule.

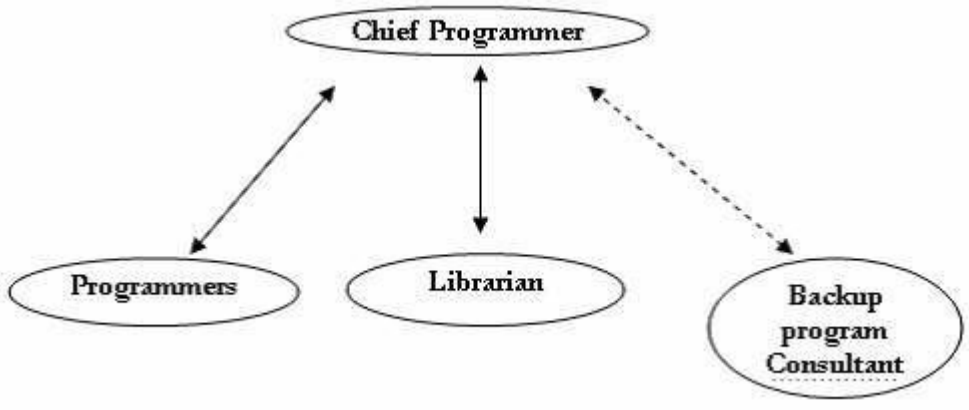

Figure: Chief Programmer Team Structure

Advantages

- Centralized decision making reduces the decision-making time.
- It reduces communication paths and related overheads.

Disadvantages

- As all the decisions are taken by the chief programmer, hence it results in low morale among the programmers.
- The effectiveness of this structure depends solely on the efficiency and knowledge of the chief programmer.

3. Hierarchical Team or Controlled Centralized (CC)

It has following features:

- It is a mixed approach of Democratic and Chief programmer team structures.
- Here the project leader has under his control, 2 to 5 senior programmers who individually have 5 to 7 junior programmers under their control.
- The various jobs of the Project leader includes
  - Assigning tasks,
  - Attending reviews and walkthroughs,
  - Detecting problem areas,
  - Balancing of the work load,
  - Participation in various technical activities.
- The major decisions are taken by the Project leader and who in-turn gives some decision making power to the senior programmers also.

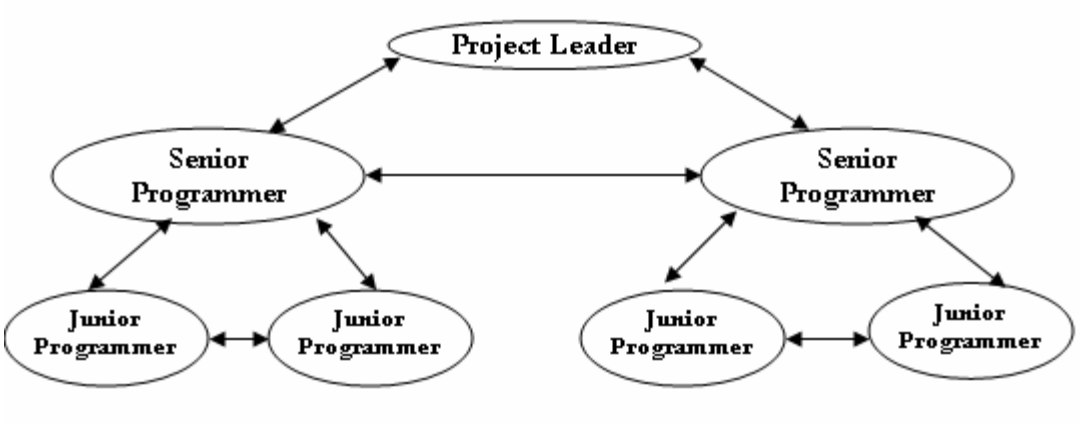

Figure: Hierarchical Team Structure

Advantages

- Here the number of communication paths are limited hence permitting effective communication
- Here the time span required for deciding and implementing the decided processes takes less time
- The job satisfaction is fairly good as the scope of promotions is good.

Disadvantages

- The most technically efficient programmers tend to be promoted, so the best programmers are lost.
- The best programmers may not be good managers hence promoted to a management post might result in reduction in productivity.

Risk Management

Risk analysis and management are a series of steps that help a software team to understand and manage uncertainty. Many problems can plague a software project. A risk is a potential problem—it might happen, it might not.

Steps of Risk Analysis and Management:

- Recognizing what can go wrong is the first step, called "risk identification."
- Each risk is analyzed to determine the likelihood that it will occur and the damage that it will do if it does occur.
- Risks are ranked, by probability and impact.
- Finally, a plan is developed to manage those risks with high probability and high impact.

In short, the four steps are Risk identification, Risk Projection, Risk assessment, and Risk management.

Risk always involves two characteristics a set of risk information sheets is produced.

- *Uncertainty*: the risk may or may not happen; that is, there are no 100% probable risks.
- *Loss*: if the risk becomes a reality, unwanted consequences or losses will occur.

Types of risks that are we likely to encounter as the software is built:

- *Project risks* threaten the project plan. That is, if project risks become real, it is likely that project schedule will slip and that costs will increase. Project risks identify potential budgetary, schedule, personnel (staffing and organization), resource, customer, and requirements problems and their impact on a software project.
- *Technical risks* threaten the quality and timeliness of the software to be produced. If a technical risk becomes a reality, implementation may become difficult or impossible.
- *Business risks* threaten the viability of the software to be built. Business risks often jeopardize the project or the product. Candidates for the top four business risks are
    - Market risk, Strategic risk, Management risk, and Budget risk.
- *Known risks* are those that can be uncovered after careful evaluation of the project plan.
- *Predictable risks* are extrapolated from past project experience (e.g., staff turnover, poor communication with the customer, dilution of staff effort as ongoing maintenance requests are serviced).
- *Unpredictable risks* are the joker in the deck. They can and do occur, but they are extremely difficult to identify in advance.

Reactive risk strategies:

- Reactive risk strategies follow that the risks have to be tackled at the time of their occurrence.
- No precautions are to be taken as per this strategy.
- They are meant for risks with relatively smaller impact.

Proactive risk strategies:

- Proactive risk strategies follow that the risks have to be identified before start of the project.
- They have to be analyzed by assessing their probability of occurrence, their impact after occurrence, and steps to be followed for its precaution.
- They are meant for risks with relatively higher impact.

Risk Identification

Risk identification is a systematic attempt to specify threats to the project plan (estimates, schedule, resource loading, etc.). By identifying known and predictable risks, the project manager takes a first step toward avoiding them when possible and controlling them when necessary.

One method for identifying risks is to create a risk item checklist. The checklist can be used for risk identification and focuses on some subset of known and predictable risks in the following generic categories:

- *Product size*: risks associated with the overall size of the software to be built or modified.
- *Business impact*: risks associated with constraints imposed by management or the marketplace.
- *Customer characteristics*: risks associated with the sophistication of the customer and the developer's ability to communicate with the customer in a timely manner.
- *Process definition*: risks associated with the degree to which the software process has been defined and is followed by the development organization.

- *Development environment*: risks associated with the availability and quality of the tools to be used to build the product.
- *Technology to be built*: risks associated with the complexity of the system to be built and the "newness" of the technology that is packaged by the system.
- *Staff size and experience*: risks associated with the overall technical and project experience of the software engineers who will do the work.

Risk Projection

Risk projection also called risk estimation, attempts to rate each risk in two ways—the likelihood or probability that the risk is real and the consequences of the problems associated with the risk, should it occur. The project planner, along with other managers and technical staff, performs four risk projection activities:

- Establish a scale that reflects the perceived likelihood of a risk,
- Delineate the consequences of the risk,
- Estimate the impact of the risk on the project and the product, and
- Note the overall accuracy of the risk projection so that there will be no misunderstandings.

Risk Assessment

At this point in the risk analysis process we have established a set of triplets of the form: $\{r_i, l_i, x_i\}$

Where, r_i is risk, l_i is the likelihood (probability) of the risk, and x_i is the impact of the risk.

During risk assessment, we further examine the accuracy of the estimates that were made during risk projection, attempt to rank the risks that have been uncovered, and begin thinking about ways to control and/or avert risks that are likely to occur.

Therefore, during risk assessment, we perform the following steps:

- Define the risk referent levels for the project.
- Attempt to develop a relationship between each (ri, li, xi) and each of the referent levels.
- Predict the set of referent points that define a region of termination, bounded by a curve or areas of uncertainty.
- Try to predict how compound combinations of risks will affect a referent level.

Risk Mitigation, Monitoring, and Management

- An effective strategy must consider three issues:
    - Risk avoidance
    - Risk monitoring
    - Risk management and contingency planning
- High staff turnover in any organization will have a critical impact on project cost and schedule.
- To mitigate the risk, project management must develop a strategy for reducing turnover. Among the possible steps to be taken are
    - Meet with current staff to determine causes for turnover.
    - Mitigate those causes that are under our control before the project starts.

- Once the project commences, assume turnover will occur and develop techniques to ensure continuity when people leave.
- Organize project teams so that information about each development activity is widely dispersed.
- Define documentation standards and establish mechanisms to be sure that documents are developed in a timely manner.
- Conduct peer reviews of all work
- Assign a backup staff member for every critical technologist.

Software Metrics

Software metrics refers to broad range of measures for computer software within the context of software project management. We are concerned with:

a) Productivity metrics output.

b) Quality metrics (customer requirements being satisfied or not)

c) Technical metrics (logically complexity of software, degree of modularity)

Why is software measured? Many reasons are there:

a) To indicate quality of product.

b) To access the productivity of people who produce the product.

c) To assess benefits (in terms of quality and productivity) derived from new software engineering methods and tools.

d) To form a baseline for estimation of cost, of resources and of schedules.

e) To help justify requests for new tools or additional training.

Types of Measures:

There are two types of measures:

1. Direct Measures – In software engineering process, following are the direct measures:

   a) Lines of Code (LOC)

   b) Execution Speed

   c) Memory Size

   d) Defects reported over some set period of time.

2. Indirect Measures – In software engineering process, following are the indirect measures:

   a) Functionality

b) Quality

c) Complexity

d) Efficiency

e) Maintainability

Note that one category of metrics is:

a) Productivity Metrics – That focuses on output of software engineering process.
b) Quality Metrics – That provides an indication of how closely software conforms to implicit and explicit customer requirements.
c) Technical Metrics – That focuses on character of software. E.g., its logical complexity, degree of modularity.

But there is another category of software metrics:

a) Size oriented metrics – Size oriented metrics are used to collect direct measures of software engineering output and quality.

b) Function oriented metrics – Function oriented metrics provide indirect measures and human oriented metrics collect information about the manner in which people develop computer software.

a) **Size oriented metrics** – It focuses on the size of the software that has been produced. If a software organization maintains simple records, a table of size-oriented measures, such as shown below can be created. The table lists each software development project that has been completed over the past few years and corresponding measures for that project.

Project	LOC	Effort	$(000)	Pp. doc.	Errors	Defects	People
alpha	12,100	24	168	365	134	29	3
beta	27,200	62	440	1224	321	86	5
gamma	20,200	43	314	1050	256	64	6
•	•	•	•	•	•		
•	•	•	•	•	•		
•	•	•	•	•	•		

Figure: Size oriented metrics

It shows that for project: alpha, 12,100 LOC were developed with 24 person months of effort, at a cost of $168,000. Note that the effort and cost recorded in the table represent all software engineering activities (i.e. analysis, design, code and test) and not just coding. Furthermore, it shows that 365 pages of documentation were developed, 134 errors were recorded before software was released and 29 defects were encountered after release to the customer within first year of operation and three people worked on this project alpha.

In order to develop metrics that can be assimilated with similar metrics from other projects, we choose lines of code as our normalization value. From the rudimentary data contained in the table, a set of simple size-oriented metrics can be developed for each project:

- Errors per KLOC (thousand lines of code).
- Defects per KLOC.
- $ per LOC.
- Page of documentation per KLOC.

In addition, other interesting metrics can be computed:

- Errors per person-month.
- LOC per person-month.
- $ per page of documentation.

Size-oriented metrics are not universally accepted as the best way to measure the process of software development. Most of the controversy swirls around the use of lines of code as a key measure. Proponents of the LOC measure claim that LOC is an "artifact" of all software development projects that can be easily counted, that many existing software estimation models use LOC or KLOC as a key input, and that a large body of literature and data predicated on LOC already exists. On the other hand, opponents argue that LOC measures are programming language dependent, that they penalize well-designed but shorter programs, that they cannot easily accommodate nonprocedural languages, and that their use in estimation requires a level of detail that may be difficult to achieve (i.e., the planner must estimate the LOC to be produced long before analysis and design have been completed).

b) **Function-Oriented Metrics** – Function-oriented software metrics use a measure of the functionality delivered by the application as a normalization value. Since ‘functionality’ cannot be measured directly, it must be derived indirectly using other direct measures. Function-oriented metrics were first proposed by Albrecht, who suggested a measure called the function point. Function points are derived using an empirical relationship based on countable (direct) measures of software's information domain and assessments of software complexity.

Function points are computed by completing the table shown in Figure above. Five information domain characteristics are determined and counts are provided in the appropriate table location. Information domain values are defined in the following manner:

Number of user inputs – Each user input that provides distinct application-oriented data to the software is counted. Inputs should be distinguished from inquiries, which are counted separately.

Number of user outputs – Each user output that provides application-oriented information to the user is counted. In this context output refers to reports, screens, error messages, etc. Individual data items within a report are not counted separately.

			Weighting factor			
Measurement parameter	**Count**		**Simple**	**Average**	**Complex**	
Number of user inputs		×	3	4	6	=
Number of user outputs		×	4	5	7	=
Number of user inquiries		×	3	4	6	=
Number of files		×	7	10	15	=
Number of external interfaces		×	5	7	10	=
Count total	→					

Figure: computing Function Point

Number of user inquiries – An inquiry is defined as an on-line input that results in the generation of some immediate software response in the form of an on-line output. Each distinct inquiry is counted.

Number of files – Each logical master file (i.e., a logical grouping of data that may be one part of a large database or a separate file) is counted.

Number of external interfaces – All machine-readable interfaces (e.g., data files on storage media) that are used to transmit information to another system are counted.

Once these data have been collected, a complexity value is associated with each count. Organizations that use function point methods develop criteria for determining whether a particular entry is simple, average, or complex. Nonetheless, the determination of complexity is somewhat subjective. To compute function points (FP), the following relationship is used:

$$FP = \text{count total} * [0.65 + 0.01 * \Sigma(F_i)]$$

where, count total is the sum of all FP entries obtained from Figure above.

The F_i (i = 1 to 14) are "complexity adjustment values" based on responses to the following questions:

1. Does the system require reliable backup and recovery?

2. Are data communications required?

3. Are there distributed processing functions?

4. Is performance critical?

5. Will the system run in an existing, heavily utilized operational environment?

6. Does the system require on-line data entry?

7. Does the on-line data entry require the input transaction to be built over multiple screens or operations?

8. Are the master files updated on-line?

9. Are the inputs, outputs, files, or inquiries complex?

10. Is the internal processing complex?

11. Is the code designed to be reusable?

12. Are conversion and installation included in the design?

13. Is the system designed for multiple installations in different organizations?

14. Is the application designed to facilitate change and ease of use by the user?

Each of these questions is answered using a scale that ranges from 0 (not important or applicable) to 5 (absolutely essential). The constant values in Equation and the weighting factors that are applied to information domain counts are determined empirically. Once function points have been calculated, they are used in a manner analogous to LOC as a way to normalize measures for software productivity, quality, and other attributes:

- Errors per FP.
- Defects per FP.
- $ per FP.
- Pages of documentation per FP.
- FP per person-month.

Advantages of FP

1. FP is programming language independent, making it ideal for applications using conventional and nonprocedural languages.

2. It is based on data that are more likely to be known early in the evolution of a project, making FP more attractive as an estimation approach.

Disadvantages of FP

1. The method requires some "sleight of hand" in that computation is based on subjective rather than objective data.

2. The counts of the information domain (and other dimensions) can be difficult to collect after the fact.

3. FP has no direct physical meaning—it's just a number.

Metrics for Source Code

Halstead's Theory: Halstead software science assigns quantitative laws to the development of computer software, using a set of primitive measures that may be derived after code is generated or estimated once design is complete.

n_1 = The number of distinct operators that appear in a program

n_2 = The number of distinct operands that appear in a program

N_1 = The total number of operator occurrences.

N_2 = The total number of operand occurrences.

Halstead uses these primitive measures to develop expressions for the overall program length, potential minimum volume for an algorithm, the actual volume (number of bits required to specify a program), the program level (a measure of software complexity), the language level (a constant for a given language) and other features such as development effort, development time and the projected number of faults in the software.

Halstead shows that length N can be estimated using the formula given below:

$$N = n_1 \log_2 n_1 + n_2 \log_2 n_2$$

And program volume may be defined as follows:

$$V = N \log_2 (n_1 + n_2)$$

where $N = N_1 + N_2$.

It should be noted that V will vary with programming language and represents the volume of information (in bits) required to specify a program.

Theoretically, a minimum volume must exist for a particular algorithm. Halstead defines a volume of the most compact form of a program to the volume of the actual program. In addition, L must always be less than 1. In terms of premature measures, the volume ratio may be expressed as

$$L = 2/n_1 * n_1/n_2$$

Information Flow Metrics (IF – Metrics)

The IF – metrics is based on a premise that all systems (software) consist of components only that work together to influence the complexity of a system. System theory tells us that components that are highly coupled and that lack cohesion tend to be less reliable and less maintainable than those that are loosely coupled and are cohesive.

IF – metrics model the degree of cohesion and coupling for a particular system component. IF – metrics was applied to software systems by Henry and Kafura. They looked at UNIX OS and found a strong association between the IF – metrics and the level of maintainability applied to the components by developers.

IF – metrics are applied to the components of a system design. For e.g., for a component/ module – A, we can define 3 measures –

1. FAN – IN: A count of the number of other components that can call or pass control, to component – A.

2. FAN – OUT: It is the number of components that are called by component – A.

3. IF – metrics of component – A is

$$IF\ (A) = [FAN - IN\ (A) * FAN - OUT\ (A)]$$

Q 1. Computer the FP value for a project with the following information domain characteristics No. of user inputs = 32

No. of user outputs = 60

No. of user inquiries = 24

No. of user files = 08

No. of External interfaces = 2

Assume that all complexity adjustment values are average. Assume 14 algorithms have been counted.

Sol: Given

No. of user inputs = 32

No. of user outputs = 60

No. of user inquiries = 24

No. of user files = 08

No. of External interfaces = 2

And complexity adjustment values are average, so,

Measurement Parameter	Count	Weighting Factor
No. of user inputs	32 * 4 =	128
No. of user outputs	60 * 5 =	300
No. of user inquiries	24 * 4 =	96
No. of user files	08 * 10 =	80
No. of External interfaces	2 * 7 =	14
Count total		618

So, FP = count total * $[0.65 + 0.01 * \Sigma(F_i)]$

$\Sigma(F_i) = 14 * 3 = 42$ (because in the question it is given that complexity adjustment values are average and all 14 algorithms are counted)

FP = 618 * [0.65 + 0.01 * 42]

FP = 661

Q 2. Computer the FP value for a project with the following information domain characteristics:

Measurement Factor		Weighting Factor
No. of user inputs	40	4
No. of user outputs	50	5
No. of user inquiries	30	5
No. of user files	10	6
No. of External interfaces	5	10

Assume that all complexity adjustment values are average i.e. have a value of 3.

Q 3. Compute FP for the following data set:

Inputs = 8

Outputs = 12

Inquiries = 4

Logical Files = 41

Interfaces = 1

$\Sigma(F_i)$ = 41 (influence factor sum).