SOFTWARE ENGINEERING (CA725)

Dr. Ghanshyam S. Bopche
Assistant Professor
Dept. of Computer Applications

September 9, 2021

SOFTWARE ENGINEERING (CA725)

Syllabus

- **Unit-I**: Introductory concepts, The evolving role of software, Its characteristics, components and applications, A layered technology, The software process, Software process models, Software Development Life cycle, Software process and project metrics, Measures, Metrics and Indicators, Ethics for software engineers.
- **Unit-II**: Software Project Planning, Project planning objectives, Project estimation, Decomposition techniques, Empirical estimation models, System Engineering, Risk management, Software contract management, Procurement Management.
- **Unit-III**: Analysis and Design, Design concept and Principles, Methods for traditional, Real time of object oriented systems, Comparisons, Metrics, Quality assurance

SOFTWARE ENGINEERING (CA725) (cont.)

- **Unit-IV**: Testing fundamentals, Test case design, White box testing, Basis path testing, Control structure testing, Black box testing, Strategies: Unit testing, integration testing, Validation Testing, System testing, Art of debugging, Metrics, Testing tools
- **Unit-V**: Formal Methods, Clean-room Software Engineering, Software reuse, Re-engineering, Reverse Engineering, Standards for industry.
- **References**:
  1. Rajib Mall, "Fundamentals of Software Engineering", 4th Edition, PHI, 2014.
  2. Roger S. Pressman, "Software Engineering-A practitioner's approach", 7 th Edition, McGraw Hill, 2010.
  3. Ian Sommerville, "Software engineering", 10th Edition, Pearson education Asia, 2016.

SOFTWARE ENGINEERING (CA725) (cont.)

4. Pankaj Jalote, “An Integrated Approach to Software Engineering”, Springer Verlag, 1997.
5. James F Peters, Witold Pedrycz, “Software Engineering – An Engineering Approach”, John Wiley and Sons, 2000.
6. Ali Behforooz, Frederick J Hudson, “Software Engineering Fundamentals”, Oxford University Press, 2009.
7. Bob Emery , “Fundamentals of Contract and Commercial Management”,Van Haren Publishing, Zaltbommel, 2013

Software Process and Project Metrics

"You can't manage what you can't measure." ©Lord Kelvin

- **Measurement**
    - Can be applied to software process with the intent of improving it on a continuous basis.
    - Can be used throughout the software project to assist in estimation, quality control, productivity assessment, and project control.
    - Can be used by software engineers to help assess the quality of work products and to assist in tactical decision making as a project proceeds.

Metrics in the Process and Project Domains

- **Process Metrics**
  - Provide a set of process indicators that lead to long-term software process improvement.
- **Project Metrics**
  Enable software project manager to
  - assess the status of an ongoing project,
  - track potential risks,
  - uncover problem areas before they go "critical",
  - adjust work flow or tasks, and
  - evaluate the project team's ability to control quality of software work products.

Process Metrics and Software Process Improvement

- Software process metrics: provide significant benefit as an organization works to improve its overall level of process maturity.
- **Rational way to improve a process**
  - Measure specific attributes of the process,
  - Develop a set of meaningful metrics based on these attributes,
  - Use the metrics to provide indicators that will lead to a strategy for improvement.

Process Metrics and Software Process Improvement (cont.)

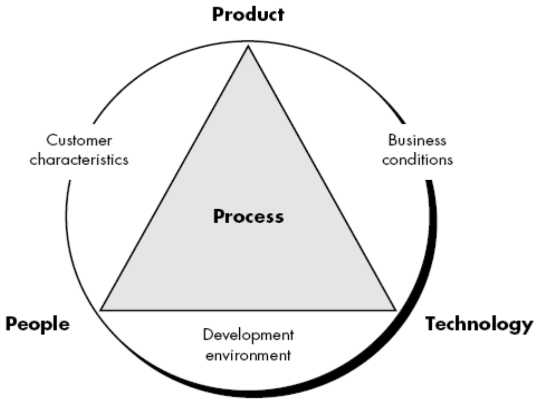


Figure: Determinants for software quality and organizational effectiveness.

Process Metrics and Software Process Improvement (cont.)

Private and public use of process data

- **Private Metrics**
  - Private to the individual software engineer.
  - Indicator for the individual only.
  - **Examples** - defect rates (by individual), defect rates (by components), and errors found during development.
- **Metrics that are private to the software project team but public to all team members**
  - **Examples** - defects reported for major software functions (that have been developed by a number of practitioners), errors found during the technical reviews, and line of code or function points per component or function.
  - Enable the improvement of team performance.
- **Public Metrics**
  - E.g. Project-level defect rates, effort, calender times, etc.
  - Enable improvement of organizational process performance.

Project Metrics

- Used by a project manager and a software team to adapt project workflow and technical activities.
- Used for effort and time estimation of a current software project.
- As a project proceeds, measures of effort and calender time expended are compared to the original estimates (and the project schedule).
- **Project metrics**
  - Production rate (represented in terms of models created, review hours, function points, and delivered source lines)
  - Errors uncovered during each software engineering task.

Project Metrics (cont.)

Intent of Project Metrics

- Project metrics are used to minimize the development schedule by making the adjustments necessary to avoid delays and mitigate potential problems and risks.
- Project metrics are used to assess product quality on an ongoing basis and, when necessary, modify the technical approach to improve quality.

Quality improvement $\rightarrow$ defects are minimized (defect count goes down) $\rightarrow$ reduced amount of rework during the project $\rightarrow$ reduced project cost

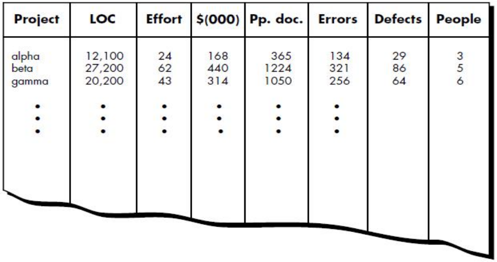
Software Measurement

- Categories of Software metrics
  1. **Direct Measure**
     - Direct measure of software process: cost and effort applied.
     - Direct measure of the product: lines of code (LOC) produced, execution speed, memory size, and defects reported over some set period of time.
     - Direct measures are easy to collect.
  2. **Indirect Measure**
     - Indirect measures of the product: functionality, quality, complexity, efficiency, reliability, maintainability, etc.
     - More difficult to assess.

Size-oriented Metrics

- Derived by normalizing quality and/or productivity measures by considering the size of the software that has been produced.
- Simple size-oriented metrics (uses lines of code as a normalization value):
  - Errors per KLOC (thousand lines of code)
  - Defects per KLOC
  - $ per KLOC
  - Pages of documentation per KLOC

Size-oriented Metrics (cont.)



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

- **Other size-oriented metrics**
  - Errors per person-month
  - KLOC per person-month
  - $ per page of documentation
- Size-oriented metrics are not universally accepted.
- LOC-based measures are programming language dependent.
  - When productivity is considered, well-designed but shorter programs are penalized.

Function-oriented Metrics

- Use a measure of the functionality delivered by the application as a normalization value.
- **Function point (FP)**
  - Most widely used function-oriented metric.
  - Computation is based on characteristics of the softwares information domain and complexity.
  - FP is programming language independent.
  - Ideal for applications using conventional and nonprocedural languages.

Function points and LOC-based metrics

- LOC and FP measures are often used to derive productivity metrics.
- Relatively accurate predictors of software development effort and cost.
- Conventional software project metrics (LOC or FP) can be used to estimate object-oriented software projects.
- These metrics do not provide enough granularity for the schedule and effort adjustments that are required as developer iterate through an evolutionary or incremental process.

Object-oriented Metrics

1. **Number of scenario scripts**
   - A scenario script (analogous to use case) is a detailed sequence of steps that describe the interaction between the user and the application.
   - Each script is organized into triplets of the form {**initiator**, *action*, **participant**} where
     - **initiator** → the object that requests some service (that initiates a message).
     - **action** → the result of the action.
     - **participant** → server object that satisfies the request.
   - The number of scenario scripts is directly correlated to the size of the application and to the number of test cases that must be developed to exercise the system once it is constructed.

Object-oriented Metrics (cont.)

2. **Number of Key Classes**
    - "Highly independent components" that are defined early in object-oriented analysis.
    - Key classes are central to the problem domain.
        - The number of key classes is an indicator of the amount of effort required to develop the software and also an indication of the potential amount of reuse to be applied during system development.

Object-oriented Metrics (cont.)

3. **Number of support classes**
   - Support classes are required to implement the system but are not immediately related to the problem domain.
   - **Examples** - user interface classes (GUI), database access and manipulation classes, and computation classes.
   - Support classes can be developed for each of the key classes.
   - Support classes are defined iteratively throughout an evolutionary process.
4. **Average number of support classes per key class**
   - If the average number of support classes per key class were known for a given problem domain, estimating (based on the total number of classes) would be greatly simplified.

Object-oriented Metrics (cont.)

5. **Number of Subsystems**
   - **Subsystem**: an aggregation of classes that support a function that is visible to the end user of system.
   - Once subsystems are identified, it is easier to lay out a reasonable schedule in which work on subsystems is partitioned among project staff.

Use Case-oriented Metrics

- Use case: a typical sequence of actions that user performs in order to complete a given task.
- Use case can help to define the scope of the system – that is, what the system must do and does not have to do.
- Use cases are used widely as a method for describing customer-level or business domain requirements that imply software features and functions.

Use Case-oriented Metrics (cont.)

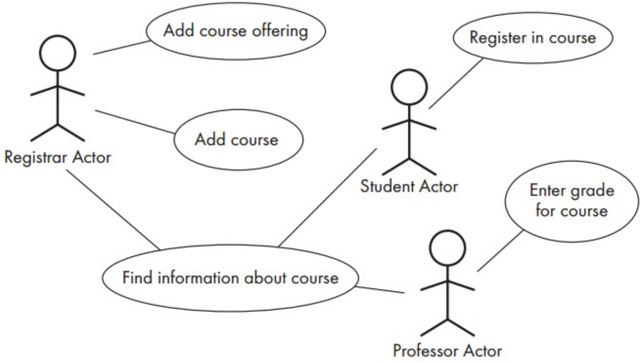


Figure: A simple use case diagram with three actors and five use cases

Use Case-oriented Metrics (cont.)

- Use case describe user-visible functions and features that are basic requirements for a system.
- As the use case is defined early in the software process, allowing it to be used for estimation before significant modeling and construction activities are initiated.
- The number of use cases is directly proportional to
  - the size of the application in LOC and
  - the number of test cases that will have to be designed to fully exercise the application.
- The use case is independent of programming language.
- Use-case points (UCPs): a mechanism for estimating project effort and other characteristics.

WebApp Project Metrics

- **WebApp Projects**: objective is to deliver a combination of content and functionality to the end user.
- **Motivation**: Measures and metrics used for traditional software engineering projects are difficult to translate directly to WebApps.

1. **Number of static Web pages**
   - Static web pages: relatively less complex and require less effort to construct than dynamic pages.
   - Provides an indication of the overall size of the application and the effort required to develop it.

WebApp Project Metrics (cont.)

2. **Number of dynamic Web pages**
   - Dynamic Web pages: represent higher relative complexity and require more effort to construct than static pages.
   - Provides an indication of the overall size of the application and the effort required to develop it.
3. **Number of internal page links**
   - Internal page links are pointers that provide a hyperlink to some other Web pages within the WebApp.
   - Provides an indication of the degree of architectural coupling within the WebApp.
   - As the number of page link increases, the effort expended on navigational decision and construction also increases.

WebApp Project Metrics (cont.)

4. **Number of persistent data objects**
   - One or more persistent data objects (e.g., a database or data file) may be accesses by a WebApp.
   - As the number of persistent data objects grows, the complexity of the WebApp also grows and the effort to implement it increases proportionally.
5. **Number of external systems interfaced**
   - WebApp must often interface with "backroom" business application.
   - As the requirement for interfacing grows, system complexity and development effort also increases.

WebApp Project Metrics (cont.)

6. **Number of static content objects**
    - Static content objects: encompass static text-based, graphical, video, animation, and audio information that are incorporated within the WebApp.
    - Multiple content objects may appear on a single Web page.
7. **Number of dynamic content objects**
    - Dynamic content objects: generated based on the end-user actions and encompass internally generated text-based, graphical, video, animation, and audio information that are incorporated within the WebApp.
    - Multiple content objects may appear on a single Web page.

WebApp Project Metrics (cont.)

8. **Number of executable functions**
   - An executable function (e.g., a script or applet) provides some computational service to the end user.
   - As the number of executable functions increases, modeling and construction effort also increases.

WebApp Project Metrics (cont.)

9. **Customization Index**
   - Reflects the degree of end-user customization that is required for the WebApp.
   - One can correlate customization index to the effort expended on the project and/or the errors uncovered as reviews and testing are conducted.
     N_{SP} = Number of static Web pages
     N_{DP} = Number of dynamic Web pages, then

$$\mathit{Customization\ Index,\ (C)} = \frac{N_{DP}}{N_{DP} + N_{SP}}$$

   - The value of C ranges from 0 to 1.
   - As C grows larger, the level of WebApp customization becomes a significant technical issue.

Metrics for Software Quality

1. **Correctness**
   - A program must operate correctly or it provide little value to its users.
   - Correctness is the degree to which the software performs its required functions.
   - Defects per KLOC - most common measure of correctness.

Metrics for Software Quality (cont.)

2. **Maintainability**
   - Maintainability is the ease with which a program can be corrected if an error is encountered, adapted if its environment changes, or enhanced if the customer desires a change in requirements.
   - There is no way to measure maintainability directly.
   - A simple time-oriented metric is *mean-time-to-change (MTTC)*.
     - The time it takes to analyze the change request, design an appropriate modification, implement the change, test it, and distribute the change to all users.
   - On average, programs that are maintainable will have a lower MTTC (for equivalent types of changes) than programs that are not maintainable.

Metrics for Software Quality (cont.)

3. **Integrity**
   - Software integrity measures the a system's ability to withstand attacks (both accidental and intentional) to its security.
   - To measure integrity, two additional attributes must be defined: threat and security.
     - **Threat**: probability (which can be derived or estimated from empirical evidences) that an attack of a specific type will occur within a given time.
     - **Security**: probability that an attack of a specific type will be repelled.

$$Integrity = [1 - (threat \times (1 - Security))]$$

Metrics for Software Quality (cont.)

4. **Usability**
   - If the program is not easy to use, it is often doomed to failure, even if the functions that it performs are valuable.
   - Usability is an attempt to quantify ease of use.

Metrics for Software Quality (cont.)

5. **Defect Removal Efficiency (DRE)**
   - A quality metric that provides benefit at both the project and process level.
   - Focus is on the errors and defects.
   - DRE is a measure of the filtering ability of quality assurance and control actions as they are applied throughout all process framework activities.
   - When considered for a project as a whole, DRE is defined as:

$$DRE = \frac{E}{E + D} \textit{where}$$

     - E $\rightarrow$ the number of errors found before delivery of the software to the end user.
     - D $\rightarrow$ the number of defects found post delivery.
   - Ideal value for DRE is 1, i.e., no defects found in the software.