Software Design

- **Design**: process of deciding "how the requirements should be implemented using the available technology?
- Encompasses the set of principles, concepts, and practices that lead to the development of a high-quality system or product.
- Creates a representation or model of the software.
- Design model: provides details about software architecture, data structures, interfaces, and components that are necessary to implement the system.
  - Design models can be assessed for quality and improved before code is generated, tests are conducted, and end users become involved in large numbers.

Software Design (cont.)

- **Goal of Software Design Process**: produce a model or representation (of a software/system) that exhibits firmness, commodity, and delight.
  - Firmness: a program should not have any bugs that inhibit its functionality.
  - Commodity: a program should be suitable for the purpose for which it was intended.
  - Delight: the experience of using the program should be pleasurable one.

Design within the context of Software Engineering

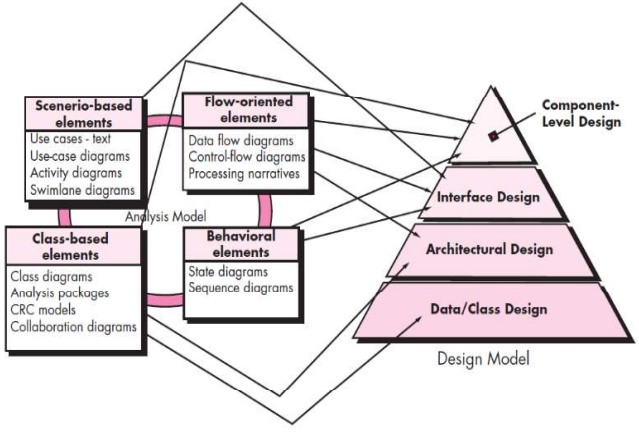


Figure: Translating the requirements model into the design model

The Design Process

- Software design: an iterative approach through which requirements are translated into a "blueprint" for constructing the software.
- Phases of software design:
  - Interface Design: the specification of the interaction between a system and its environment.
  - Architectural Design: the specification of the major components of a system, their responsibilities, properties, interfaces, and the relationships and interactions between them.
  - Detailed Design: the specification of the internal elements of all major system components, their properties, relationships, processing, and often their algorithms and the data structures.

The Design Process (cont.)

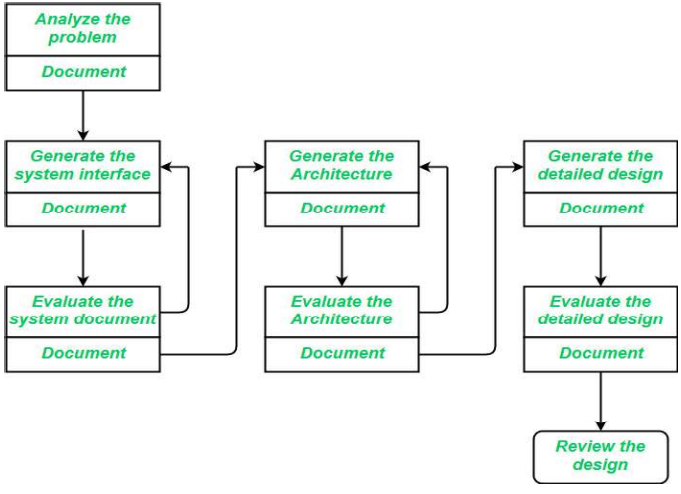


Figure: Software design Process

Characteristics for the evaluation of a good design

The design

- must implement all of the explicit requirements contained in the requirements model, and it must accommodate all of the implicit requirements desired by stakeholders.
- must be a readable, understandable guide for those who generate code and for those who test and subsequently support the software.
- should provide a complete picture of the software, addressing the data, functional, and behavioral domains from an implementation perspective.

Quality guidelines for the Software Design

A design should

- exhibit an architecture that
  - has been created using recognizable architectural styles or patterns,
  - is composed of components that exhibit good design characteristics, and
  - can be implemented in an evolutionary fashion, thereby facilitating implementation and testing.
- be modular; that is, the software should be logically partitioned into elements or subsystems.
- contain distinct representations of data, architecture, interfaces, and components.
- Lead to data structures that are appropriate for the classes to be implemented and are drawn from recognizable data patterns.

Quality guidelines for the Software Design (cont.)

- lead to components that exhibit independent functional characteristics.
- lead to interfaces that reduce the complexity of connections between components and with the external environment.
- be derived using a repeatable method that is driven by information obtained during software requirements analysis.
- be represented using a notation that effectively communicates its meaning.

Software Design Quality Attributes

- FURPS quality attributes (Functionality, Usability, Reliability, Performance, Supportability)
  - Developed by Hewlett-Packard.
- Functionality: assessed by evaluating the feature set and capabilities of the program, the generality of the functions that are delivered, and the security of the overall system.
- Usability: assessed by considering human factors, overall aesthetics, consistency, and documentation.
- Reliability: evaluated by measuring the frequency and severity of failure, the accuracy of output results, the mean-time-to-failure (MTTF), the ability to recover from failure, and the predictability of the program.

Software Design Quality Attributes (cont.)

- Performance: measured by considering processing speed, response time, resource consumption, throughput, and efficiency.
- Supportability
  - combines
    - the ability to extend the program (extensibility),
    - adaptability,
    - serviceability.

    Three attributes represent a more common term, maintainability.
  - In addition, supportability refers to
    - testability,
    - compatibility,
    - configurability (the ability to organize and control elements of the software configuration),
    - the ease with which a system can be installed, and
    - the ease with which problems can be localized.

Design Concepts

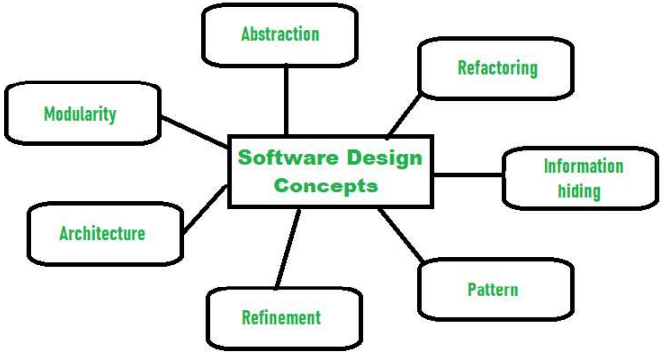


Figure: Software design concepts

Design Concepts (cont.)

Each design concept help software engineer to answer following questions:

- What criteria can be used to partition software into individual components?
- How is function or data structure detail separated from a conceptual representation of the software?
- What uniform criteria define the technical quality of a software design?

Design Concepts (cont.)

1. **Abstraction**
   - Modular solution (to any problem)
     - Need abstraction at many levels.
     - Highest level of abstraction: a solution is stated in broad terms using the language of the problem environment.
     - Lower level of abstraction: a more detailed description of the solution is provided.
   - Procedural abstraction
     - Refers to a sequence of instructions that have a limited and specific function.
     - Example: the word **open** for a door.
   - Data abstraction
     - A named collection of data that describes a data object.
     - Example: In the context of the procedural abstraction **open**, we can define a data abstraction called **door.**

Design Concepts (cont.)

2. **Architecture**
   - Rrefers to "the overall structure of the software and the ways in which that structure provide conceptual integrity for a system".
   - The structure or organization of program components (modules), the manner in which these components interact, and the structure of data that are used by the components.

Design Concepts (cont.)

- The architectural design can be represented using one or more number of different models:
  - 2.1 Structural Models: represent architecture as an organized collection of program components.
  - 2.2 Framework Models: increase the level of design abstraction by attempting to identify repeatable architectural design frameworks that are encountered in similar types of applications.
  - 2.3 Dynamic Models: address the behavioral aspects of the program architecture, indicating how the structure or system configuration may change as a function of external events.
  - 2.4 Process Models: focus on the design of the business or technical process that the system must accommodate.
  - 2.5 Functional Models: used to represent the functional hierarchy of a system.

Design Concepts (cont.)

3. **Patterns**
   - A design pattern is a design structure that solves a particular design problem within a specific context and amid "forces" that may have an impact on the manner in which the pattern is applied and used.
   - The intent of each design pattern is to provide a description that enables a designer to determine
     - whether the pattern is applicable to the current work,
     - whether the pattern can be reused (hence, saving design time), and
     - whether the pattern can serve as a guide for developing a similar, but functionally of structurally different pattern.

Design Concepts (cont.)

4. **Separation of Concerns**
   - A design concept that suggests that any complex problem can be more easily handled if it is subdivided into pieces that can each be solved and/or optimized independently.
   - A *concern* is a feature or behavior that is specified as part of the requirements model for the software.
   - By separating concerns into smaller, and therefore more manageable pieces, a problem takes less effort and time to solve.

Design Concepts (cont.)

- The perceived complexity of two problems when they are combined is often greater than the sum of the perceived complexity when each is taken separately, i.e.,

$$Complexity(P_1 + P_2) \geq Complexity(P_1) + Complexity(P_2)$$

- Divide-and-conquer strategy: it is easier to solve a complex problem when you break it into manageable pieces.
- Separation of concerns is manifested in following related design concepts:
  - modularity,
  - aspects,
  - functional independence, and
  - refinement.

Design Concepts (cont.)

5. **Modularity**
   - Most common manifestation of separation of concerns.
   - Software is divided into separately named and addressable components, sometimes called *modules*, that are integrated to satisfy problem requirements.
   - Modularity allows a program to be intellectually manageable.

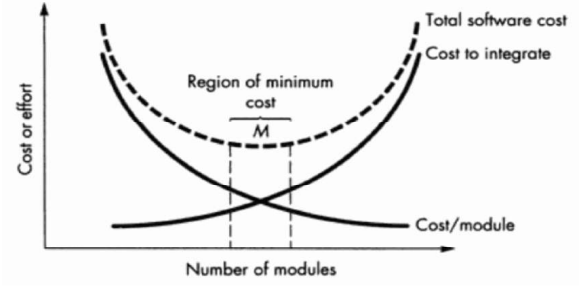


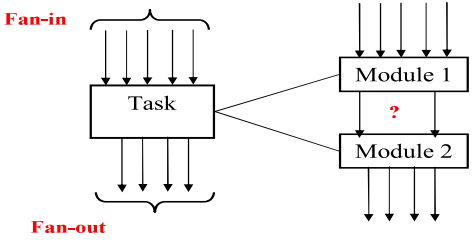
Figure: Modularity and software cost

Design Concepts (cont.)

- **Fundamental Question**: "How do we decompose a software solution to obtain the best set of modules?"
- **Answer**: modularize a design (and the resulting program) so that
  - the development can be easily planned,
  - software increments can be defined and delivered,
  - changes can be more easily accommodated,
  - testing and debugging can be conducted more efficiently, and
  - long-term maintenance can be conducted without serious side effects.
- Consider two subprograms, A and B
  - Effort (A + B) $\geq$ Effort(A) + Effort(B)
  - Cost (A + B) $\geq$ Cost(A) + Cost(B)

Design Concepts (cont.)

Fan-in Fan-out Formula (Henry and Katura Formula)



$$L((Fan-in) \times (Fan-out))^2$$

$$L(5 \times 4)^2 = \frac{L}{2}(5 \times x)^2 + \frac{L}{2}(x \times 4)^2$$

$$400 = \frac{25x^2 + 16x^2}{2} \Rightarrow x = \sqrt[2]{\frac{800}{41}} = 4.41 \equiv 4$$

Design Concepts (cont.)

6. **Information Hiding**
   - Principle of information hiding: modules must be characterized by design decision that (each) hides from all others.
   - Modules must be specified and designed so that information (algorithms and data) contained within a module is inaccessible to other modules that have no need of such information.
   - Hiding implies that effective modularity can be achieved by defining a set of independent modules that communicate with one another only that information necessary to achieve software function.

Design Concepts (cont.)

- Hiding defines and enforces access constraints to both procedural detail within a module and any local data structure used by the module.
- **Benefits of Information Hiding**
  - Information hiding provides the greatest benefits when modifications are required during testing and later during software maintenance.
  - Because most data and procedural detail are hidden from other parts of the software, inadvertent errors introduced during modification are less likely to propagate to other locations within the software.

Design Concepts (cont.)

7. **Functional Independence**
   - Direct outgrowth of separation of concerns, modularity, and the concepts of abstraction and information hiding.
   - Achieved by developing modules with "single minded" function and an "aversion" to excessive interaction with other modules.
   - Software must be designed in such a way that each module addresses a specific subset of requirements and has a simple interface when viewed from other parts of the program structure.

Design Concepts (cont.)

- **Benefits of functional independence**
  - Software with effective modularity (i.e., independent modules) is easy to develop.
  - Independent modules are easier to maintain (and test).
    $\rightarrow$ Secondary effects caused by design or code modification are limited, error propagation is reduced, and reusable modules are possible.
- Functional independence is a key to good design, and design is the key to software quality.

Design Concepts (cont.)

- **Qualitative criteria to assess functional independence**: (a) cohesion, and (b) coupling.
  (a) Cohesion
  - Cohesion is a measure of the relative functional strength of a module.
  - Natural extension of the information hiding concept.
  - A cohesive module performs a single task (just do one thing), requiring little interaction with other components in other parts of a program.
  - As per the goal of modularity, cohesion should be high.

Design Concepts (cont.)

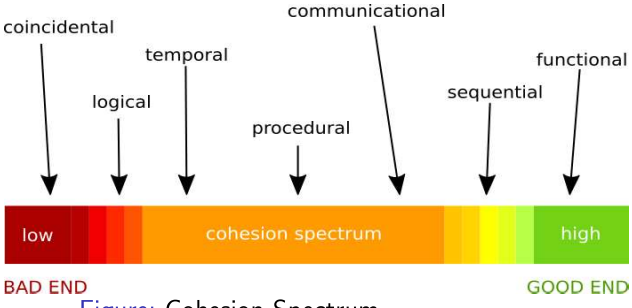


Figure: Cohesion Spectrum

- **Coincidental Cohesion**: if the elements in the module are unrelated.
- **Logical Cohesion**: if the elements of the module are related and they perform set of operations.
- **Temporal Cohesion**: if the elements of the module are related and they are confined to initialization and time.

Design Concepts (cont.)

- **Procedural Cohesion**: if the elements in the module are related and they are confined to one name such that it performs collection of operations.
- **Communicational Cohesion**: if the elements in the module are related and if they interact through data declared in the module.
- **Sequential Cohesion**: if the elements in the module are related and if they perform set of operations such that output of one operation will be the input of other operation.
- **Functional Cohesion**: if the elements in the module are related and the module perform one and only one operation.

Design Concepts (cont.)

(b) Coupling

- Coupling is a measure of the relative interdependence among modules.
- Coupling depends on the interface complexity between modules, the point at which entry or reference is made to a module, and what data pass across the interface.
- As per the goal of modularity coupling should be low.
- Simple connectivity among modules results in software that is easier to understand and less prone to a "ripple effect", caused when errors occur at one location and propagate throughout a system.

Design Concepts (cont.)

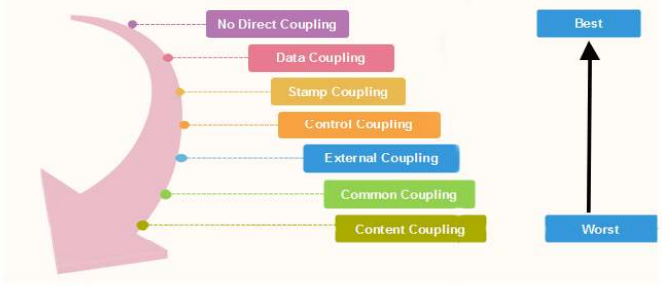


Figure: Coupling Spectrum

- **Procedural call coupling**: a form of coupling in which interaction between the modules is nominal i.e., all modules are independent.
- **Low coupling**: a coupling in which interaction between the modules is minimum in extremity no coupling between them.
- **Inclusion coupling**: a coupling in which source code of one module is included in another module.

Design Concepts (cont.)

- **Import coupling**: a coupling in which one module is declared in another module.
- **External coupling**: a form of coupling which represents interaction between internal module of a project with external modules developed by third party. It may be specific software or hardware.
- **Data coupling**: a coupling in which modules interact using simple or homogeneous data i.e., parameters, variables, so on.
- **Stamp coupling**: a coupling in which modules interact through composite and heterogeneous data i.e., structures, unions, graphs, trees, linked list and so on.
- **Control coupling**: a coupling in which one module control the order of execution of other modules using flags.
  This type of communication is also referred as parent-child or server-client or base class-derived class or superclass-subclass communication.

Design Concepts (cont.)

- **Common coupling**: a form of coupling in which modules interact through common sharable resource.
  This resource can be software component or hardware component.
- **Content coupling**: a coupling in which module refers to another module, in extremity it changes the internal structure of another module.

Design Concepts (cont.)

8. **Refinement**
   - Stepwise refinement is a top-down design strategy.
   - A program is developed by successively refining levels of procedural detail.
   - A hierarchy is developed by decomposing a macroscopic statement of function (a procedural abstraction) in a stepwise fashion until programming language statements are reached.
   - Refinement is actually a process of elaboration.

Design Concepts (cont.)

- Abstraction and refinement are complementary concepts.
  - Abstraction enables you to specify procedure and data internally but suppress the need for "outsiders" to have knowledge of low-level details.
  - Refinement helps you to reveal low-level details as design progresses.
  - Both concepts allow designer to create a complete design model as the design evolves.

Design Concepts (cont.)

9. **Aspects**
   - Concerns are uncovered during requirements analysis.
   - Concerns include requirements, use cases, features, data structures, quality-of-service issues, variants, intellectual property boundaries, collaborations, patterns and contracts.
   - Ideally, a requirements model can be organized in a way that allows you to isolate each concern (requirement) so that it can be considered independent.

Design Concepts (cont.)

- An aspect is a representation of a crosscutting concern.
  - Consider two requirements, A and B.
    Requirement A crosscuts requirement B "if a software decomposition (refinement) has been chosen in which B cannot be satisfied without taking A into account."
- It is important to identify aspects so that the design can properly accommodate them as refinement and modularization occur.
- In an ideal context, an aspect is implemented as a separate module (component).

Design Concepts (cont.)

10. **Refactoring**
    - An important design activity suggested for many agile methods.
    - Refactoring is a reorganization technique that simplifies the design (or code) of a component without changing its function or behavior.
    - Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure.
    - When software is refactored, the existing design is examined for redundancy, unused design elements, inefficient or unnecessary algorithms, poorly constructed or inappropriate data structures, or any other design failure that can be corrected to yield a better design.

Design Concepts (cont.)

11. Object-Oriented Design Concepts

- The object-oriented (OO) paradigm is widely used in modern software engineering.

11.1 Class

- Units of data abstraction in an object-oriented program.
- A class is a template, blueprint, or contract that defines what an object's data fields and methods will be.

11.2 Object

- An object is an instance of a class. One can create many instances of a class.
- All the objects with the same properties and behavior are instances of one class.

Design Concepts (cont.)

11.3 Attribute

- A simple piece of data used to represent the properties of an object.
- For example, each instance of class Employee might have the attributes: name, dateOfBirth, socialSecurityNumber, telephoneNumber, address.

11.4 Association

- Represents the relationship between instances of one class and instances of another.
- For example, class Employee in a business application might have the following relationships:
  → supervisor (association to class Manager)
  → tasksToDo (association to class Task)

11.5 Method

- Procedural abstractions used to implement the behavior of a class.

Design Concepts (cont.)

11.6 Operation

- An operation is a higher-level procedural abstraction.
- It is used to discuss and specify a type of behavior, independently of any code that implements that behavior.
- Several different classes can have methods with the same name that implement the abstract operation in ways suitable to each class.

11.7 Polymorphism

- Polymorphism is a property of object-oriented software by which an abstract operation may be performed in different ways, typically in different classes.
- An operation is said to be polymorphic, if the running program decides, every time an operation is called, which of several identically named methods to invoke.

Design Concepts (cont.)

11.8 Inheritance

- If several classes have attributes, associations or operations in common, it is best to avoid duplication by creating a separate *superclass* that contains these common aspects.
- Inheritance allows us to derive new classes from existing classes. If you have a complex class, it may be good to divide its functionality among several specialized subclasses.
- Inheritance is an important and powerful feature of object oriented software engineering for reusing software code.
- Inheritance is the implicit possession by a subclass of features defined in a superclass. Features include variables and methods.
- The relationship between a subclass and an immediate superclass is called a *generalization*.
- The subclass is called a *specialization*.
- A hierarchy with one or more *generalizations* is called an *inheritance hierarchy*.

Design Concepts (cont.)

11.9 Method Overriding

- A subclass inherits methods from a superclass.
- Sometimes it is necessary for the subclass to modify the implementation of a method defined in the superclass. This is referred to as method overriding.

Design Concepts (cont.)

11.10 Interfaces

- An interface is a classlike construct that contains only constants and abstract methods.
- In many ways an interface is similar to an abstract class, but the intent is to specify common behavior for objects.
- Interface neither have instance variables nor concrete methods.
- Interface is a named list of abstract operations.
- Several implementing classes of an interface that must implement the abstract operations.
- In Java, a class can implement multiple interfaces, but can have only one superclass.
- **Interfaces in Java**: Comparable, ActionListener, Cloneable, Runnable, etc.

Design Concepts (cont.)

12. **Design Classes**
    - The requirements model defines a set of analysis classes.
    - Each analysis class describes some element of the problem domain, focusing on aspects of the problem that are user visible.
    - As the design model evolves, a set of design classes refine the analysis classes by providing design detail that will enable the classes to be implemented, and implement a software infrastructure that supports the business solution.

Design Concepts (cont.)

- **Types of design classes**
  - 12.1 User interface classes
    → Define all abstractions that are necessary for human computer interaction (HCI).
  - 12.2 Business domain classes
    → Refinements of the analysis classes defined earlier.
    → The classes identify the attributes and services (methods) that are required to implement some element of the business domain.
  - 12.3 Process Classes
    → Implement lower-level business abstractions required to fully manage the business domain classes.

Design Concepts (cont.)

12.4 Persistent Classes
→ represent data stores (e.g., a database) that will persist beyond the execution of the software.

12.5 System Classes
→ Implement software management and control functions that enable the system to operate and communicate within its computing environment and with the outside world.

The Design Model

Dimensions of Design Model: Process Dimension vs Abstraction Dimension

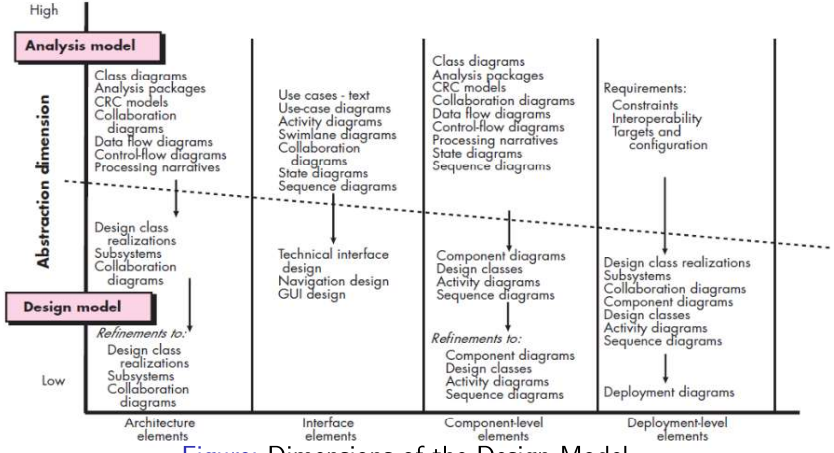


Figure: Dimensions of the Design Model

The Design Model (cont.)

1. **Data Design Elements**
   - Data design (or data architecting) creates a model of data and/or information that is represented at a high level of abstraction (the customer/user's view of data).
   - The data model is then refined into progressively more implementation-specific representations that can be processed by the computer-based system.
   - Program component level
     - The design of data structures and the associated algorithms.
   - Application level
     - The translation of a data model into a database (pivotal to achieving the business objectives of a system).
   - Business level
     - The collection of information stored in different databases and reorganized into a data warehouse (enables data mining or knowledge discovery that can have an impact on the success of the business itself).

The Design Model (cont.)

2. **Architectural Design Elements**
   - Architectural design elements provides an overall view of the software (equivalent to the floor plan of a house).
   - The architectural model is derived from three sources:
     - 2.1 Information about the application domain for the software to be built,
     - 2.2 Specific requirements model elements such as data flow diagrams or analysis classes, their relationships and collaborations for the problem at hand, and
     - 2.3 The availability of architectural styles and patterns.
   - The architectural design element is usually depicted as a set of interconnected subsystems, often derived from analysis packages within the requirements model.
   - Each subsystem may have it's own architecture.

The Design Model (cont.)

3. **Interface Design Elements**
   - Depict information flows into and out of the system and how it is communicated among the components defined as part of the architecture.
   - Analogous to a set of detailed drawings (and specifications) for the doors, windows, and external utilities of a house.
   - Elements of an interface design
     - 3.1 The user interface (UI),
     - 3.2 External interfaces to other systems, devices, networks, or other producers or consumers of information, and
     - 3.3 Internal interfaces between various design components.

The Design Model (cont.)

4. **Component-Level Design Elements**
   - Equivalent to a set of detailed drawings (and specifications) for each room in a house.
   - Fully describes the internal detail of each software component.
   - The component-level design defines data structures for all local data objects and algorithmic detail for all processing that occurs within a component and an interface that allows access to all component operations (behaviors).
5. **Deployment-Level Design Elements**
   - Indicate how software functionality and subsystems will be allocated within the physical computing environment that will support the software.

SOFTWARE ENGINEERING (CAS725)

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

October 21, 2021

SOFTWARE ENGINEERING (CAS725)

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 (CAS725) (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 (CAS725) (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 Quality Assurance

Definition: an effective software process applied in a manner that creates a useful product that provides measurable value for those who produce it and those who use it.

- An effective software process establishes the infrastructure that supports any effort at building a high-quality software.
- A useful product delivers the content, functions, and features that the end user desires, but as important, it delivers these assets in a reliable, error-free way.
- By adding value for both the producer and user of a software product, high quality software provides benefits for the software organization and the end user community.

Quality Dimensions

- Performance quality
- Feature quality
- Reliability
- Conformance
- Durability
- Serviceability
- Aesthetics
- Perception

McCall's Quality Factors

McCall's software quality factors focus on three important aspects of a software product:

- operational characteristics,
- ability to undergo change, and
- adaptability to new environments.

McCall's Quality Factors (cont.)

1. **Correctness**: the extent to which a program satisfies its specification and fulfills the customer's mission objectives.
2. **Reliability**: the extent to which a program can be expected to perform its intended function with required precision.
3. **Efficiency**: the amount of computing resources and code required by a program to perform its function.
4. **Integrity**: extent to which access to software or data by unauthorized persons can be controlled.
5. **Usability**: effort required to learn, operate, prepare input for, and interpret output of a program.

McCall's Quality Factors (cont.)

6. **Maintainability**: effort required to locate and fix an error in a program.
7. **Flexibility**: effort required to modify an operational program.
8. **Testability**: effort required to test a program to ensure that it performs its intended function.
9. **Portability**: effort required to transfer the program from one hardware and/or software system environment to another.
10. **Reusability**: extent to which a program (or parts of a program) can be reused in other applications.
11. **Interoperability**: effort required to couple one system to another.

ISO 9126 Quality Factors

1. **Functionality**: the degree to which the software satisfies stated needs as indicated by the subattributes: suitability, accuracy, interoperability, compliance, and security.
2. **Reliability**: the amount of time the software is available for use as indicated by the subattributes: maturity, fault tolerance, recoverability.
3. **Usability**: The degree to which the software is easy to use as indicated by the substtributes: understandability, learnability, operability.

ISO 9126 Quality Factors (cont.)

4. **Efficiency**: The degree to which the software makes optimal use of system resources as indicated by the following subattributes: time behavior, resource behavior.
5. **Maintainability**: The ease with which repair may be made to the software as indicated by the following subattributes: analyzability, changeability, stability, testability.
6. **Portability**: The ease with which the software can be transported from one environment to another as indicated by the following subattributes: adaptability, installability, conformance, replaceability.

The Software Quality Dilemma

- "Good Enough" Software
- The cost of quality
  - Time, and
  - Money

Software Quality Assurance (SQA)

- Goal of Software engineering: produce on-time, high-quality software.
- Software quality assurance or quality management: an umbrella activity that is applied throughout the software process.
- Software quality assurance encompasses
  - SQA process,
  - specific quality assurance and quality control tasks (including technical reviews and a multi-tiered testing strategy),
  - effective software engineering practices (methods and tools),
  - control of all software work products and the changes made to them,
  - a procedure to ensure compliance with software development standards, and
  - measurement and reporting mechanisms.

Elements of Software Quality Assurance

- Standards
  - Broad array of software engineering standards and related documents from the IEEE, ISO, and other standards organizations.
  - Role of SQA: ensure that standards that have been adopted are followed and that all work products conform to them.
- Reviews and Audits
  - Technical review - a quality control activity.
  - Goal: uncover errors.
  - Audits are a type of reviews performed by SQA personnel with the intent of ensuring that quality guidelines are being followed for software engineering work.

Elements of Software Quality Assurance (cont.)

- Testing
  - Primary goal: find errors.
  - The job of SQA is to ensure that testing is properly planned and efficiently conducted so that it has the highest likelihood of uncovering errors.
- Errors/defect collection and analysis
  - SQA team collects and analyze error and defect data to better understand "how errors are introduced and what software engineering activities are best suited to eliminate them?"

Elements of Software Quality Assurance (cont.)

- Change Management
  - Change - one of the most disruptive aspect of any software project.
  - If change is not properly managed, it can lead to confusion, and confusion almost always leads to poor quality.
  - SQA ensures that adequate change management practices have been instituted.
- Education
  - Improvement of software engineering practices through education.

Elements of Software Quality Assurance (cont.)

- Vendor Management
  The job of SQA organization is to ensure the high-quality software results by
  - suggesting specific quality practices that the vendor should follow (when possible), and
  - incorporating quality mandates as part of any contract with an external vendor.
- Security Management
  - **Concern**: Cyber crime, and privacy issues.
  - SQA ensures that appropriate process and technology are used to achieve software quality.

Elements of Software Quality Assurance (cont.)

- Safety
  - Software - a pivotal component of human-rated systems (e.g., automotive or aircraft applications).
  - The impact of hidden defects can be catastrophic.
  - SQA may be responsible for assessing the impact of software failure and for initiating those steps required to reduce risk.
- Risk Management
  - The SQA organization ensures that risk management activities are properly conducted and that risk-related contingency plan have been established.

SQA Tasks

- The Software Engineering Institute (SEI) recommends a set of SQA actions that address quality assurance planning, record keeping, analysis, and reporting.
- These actions are performed by an independent SQA group that
  - Prepares SQA plan for a project.
  - Participate in the development of the project's software process description.
  - Reviews software engineering activities to verify compliance with the defined software process.
  - Audits designated software work products to verify compliance with those defined as part of the software process.
  - Ensures that deviations in software work and work products are documented and handled according to the documented process.
  - Record any noncompliance and reports to senior management.

SQA Goals

- Requirements Quality
  - Focus is on the correctness, completeness, and consistency of the requirement model.
  - SQA must ensure that the software team has properly reviewed the requirements model to achieve a high level of quality.
- Design Quality
  - Focus is on the quality of every element of the design model.
  - SQA looks for attributes of the design that are indicators of quality.

SQA Goals (cont.)

- Code Quality
  - Source code and related work products (e.g., other descriptive information) must conform to local coding standards and exhibit characteristics that will facilitate maintainability.
  - SQA should isolate those attributes that allow a reasonable analysis of the quality of code.
- Quality control effectiveness
  - A software team should apply limited resources in a way that has the highest likelihood of achieving a high-quality result.
  - SQA analyzes the allocation of resources for reviews and testing to assess whether they are being allocated in the most effective manner.

Software quality goals, attributes, and metrics

Goals	Attributes	Metric
Requirement quality	• Ambiguity	• Number of ambiguous modifiers (e.g., many, large, human-friendly)
	• Completeness	• Number of TBAs, TBDs
	• Understandability	• Number of sections/subsections
	• Volatility	• Number of changes per requirement
		• Time (by activity) when change is requested
	• Traceability	• Number of requirements not traceable to design/code
	• Model clarity	• Number of UML models
		• Number of descriptive pages per model
		• Number of UML errors
Design quality	• Architectural integrity	• Existence of architectural model
	• Component completeness	• Number of components that trace to architectural model
		• Complexity of procedural design
	• Interface complexity	• Layout appropriateness
	• Patterns	• Number of patterns used

Software quality goals, attributes, and metrics

(cont.)

Goals	Attributes	Metric
Code quality	• Complexity	• Cyclomatic complexity
	• Maintainability	• Design factors
	• Understandability	• Percent internal comments • Variable naming conventions
	• Reusability	• Percent reused components
	• Documentation	• Readability index
QC effectiveness	• Resource allocation	• Staff hour percentage per activity
	• Completion rate	• Actual vs. budgeted completion time
	• Review effectiveness	• Review metrics
	• Testing effectiveness	• Number of errors found and criticality • Effort required to correct an error • Origin of error

Statistical Software Quality Assurance

- Reflects a growing trend throughout industry to become more quantitative about quality.
- Important steps in statistical quality assurance
  - Information about software errors and defects is collected and categorized.
  - An attempt is made to trace each error and defect to its underlying cause (e.g., non-conformance to specifications, design error, violation of standards, poor communication with the customer).
  - Using the Pareto principle (80% of the defects can be traced to 20% of all possible causes), isolate the 20% (the vital few).
  - Move to correct the problems that have caused the errors and defects.

Six Sigma for Software Engineering

- **Six Sigma**: most widely used strategy for statistical quality assurance in industry.

- **Six Sigma Strategy**: "a rigorous and disciplined methodology that uses data and statistical analysis to measure and improve a companies operational performance by identifying and eliminating defects in manufacturing and service-related processes."

Six Sigma for Software Engineering (cont.)

- Six Sigma is derived from Six standard deviations (6σ).
- 3.4 instances (defects) per million occurrences - implying extremely high quality standard.
- Core steps of Six Sigma Methodology
  - Define customer requirements and deliverables and project goals via well-defined methods of customer communication.
  - Measure the existing process and its output to determine current quality performance (collect defect metrics).
  - Analyze defect metrics and determine the vital few causes.

Six Sigma for Software Engineering (cont.)

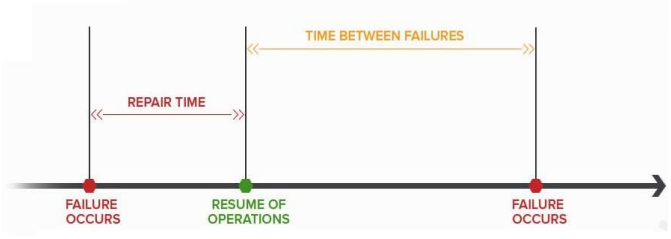
- If the existing software process is in place, but improvement is required, Six Sigma suggests Two additional steps:
  - Improve the process by eliminating the root causes of defects.
  - Control the process to ensure that future work does not reintroduce the causes of defects.
- If the organization is developing a software process (rather than improving an existing process), the core steps are augmented as follows:
  - Design the process to
    - avoid the root causes of defects and
    - to meet customer requirements.
  - Verify that the process model will, in fact, avoid defects and meet customer requirements.

Software Reliability

- Software is more reliable if it has fewer failures.
- Failure: non-conformance to software requirements.
- All software failures can be traced to design or implementation problems.
- Reliability depends on the number and type of mistakes a software engineer make.
- Software reliability can be measured directly and estimated using historical and developmental data.
- In statistical terms software reliability is defined as "the probability of failure-free operation of a computer program in a specified environment for a specified time".

Software Reliability (cont.)

Measures of Reliability and Availability



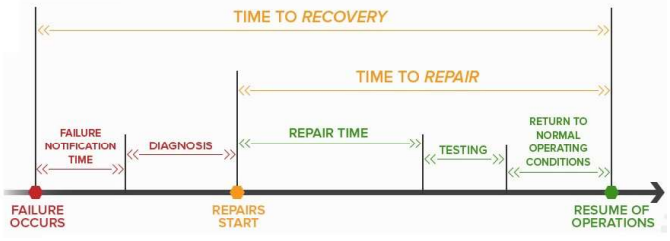
- A simple measure of reliability is mean-time-between-failure (MTBF).

$$MTBF = MTTF + MTTR$$

  where the acronyms MTTF and MTTR are mean-time-to-failure and mean-time-to-recover, respectively.

- MTBF reliability measure is equally sensitive to both MTTF and MTTR.

Software Reliability (cont.)



- MTBF can be problematic for two reasons:
  - It projects a time span between failures, but does not provide us with a projected failure rate, and
  - MTBF can be misinterpreted to mean average life span even though this is not what it implies.

Software Reliability (cont.)

- Software availability is the probability that a program is operating according to requirements at a given point in time and is defined as

$$Availability = \frac{MTTF}{MTTF + MTTR} \times 100\%$$

- Software non-availability = 1 - Availability
- The availability measure is somewhat more sensitive to MTTR (an indirect measure of the maintainability of software).

Software Reliability (cont.)

Question: A company has released a business software on which customer started to work. Customer works smoothly almost for Two and Half Years then the first bug/defect has occurred. The maintenance team has taken Four days to detect and fix the bug/defect. Calculate MTBF, availability and non-availability of the product.

Solution

- Mean-time-to-failure (MTTF) $= 2\frac{1}{2}$ Years $= 912$ Days $= 912 \times 24$ Hours $= 21888$ Hours
- Mean-time-to-recover (MTTR) $= 4$ Days $= 4 \times 24$ Hours $= 96$ Hours

Software Reliability (cont.)

- Mean-time-between-failure (MTTB) = MTTF + MTTR
  = 21984 Hours
- Software Availability = $\frac{MTTF}{MTTF+MTTR}$
  = $\frac{21888}{21888+96}$
  = $\frac{21888}{21984}$
  = 0.9956 = 99.56%
- Software Non-availability = 1 - Availability
  = 1 - 0.9956
  =0.0044 = 0.44%

The SQA Plan

- Provides a road map for instituting software quality assurance.
- The plan serves as a template for SQA activities that are instituted for each software project.
- The standards for SQA plans has been published by the IEEE, and it recommends a structure that identifies
  - The purpose and scope of the plan,
  - A description of all software engineering work products (e.g., models, documents, source code) that fall within the purview of SQA,
  - All applicable standards and practices that are applied during the software process,
  - SQA actions and tasks (including reviews and audits) and their placement throughout the software process,

The SQA Plan (cont.)

- The tools and methods that support SQA actions and tasks,
- Software configuration management procedures,
- Method for assembling, safeguarding, and maintaining all SQA-related records, and
- Organizational roles and responsibilities relative to product quality.