PGDCA

Paper Name: Software Engineering

Topic: Software Design

Paper Code: PGDCA C206

SEMESTER II

By Ms. Amrita Prakash

Assistant Professor,

Department of Computer Science

Patna Women's College

E-mail: amrita.bca@patnawomenscollege.in

Software Design

Software design is a software engineering activity where software requirements are analyzed in order to produce a description of the internal structure and organization of the system that serves as a basis for its construction (coding).

IEEE defines software design as "*both a process of defining the architecture, components, interfaces, and other characteristics of a system or component and the result of that process*".

During software design phase, many critical and strategic decisions are made to meet the required functional and quality requirements of a system. These decisions are taken into account to successfully develop the software and carry out its maintenance in a systematic manner to improve the quality of the end product.

Objectives of Software Design:

The main objective of software design phase is to develop a blue print which serves as a base while developing the software system. The other objectives of software design are listed below:

- To produce various models that can be analyzed and evaluated to determine if they will allow the various requirements to be fulfilled.
- To examine and evaluate various alternative solutions and trade-offs involved.
- To plan subsequent software development activities.

Principles of Software Design

Developing design is a cumbersome process as most expansive errors are often introduced in this phase. Since problems in the design phase can be very expensive to solve in later stages of the software development, a number of principles are considered while designing the software. These principles act as a framework for the designers to follow a good design practice. Some of the commonly followed design principles are listed below:

- **Software design should be traceable to the analysis model:** As a single design element often relates to multiple requirements, it becomes essential to have a means for tracking how requirements are satisfied by the design model.

- **Choose the right programming paradigm:** A programming paradigm is the framework used for designing and describing the structure of the software

system. The two most popular programming paradigms are the procedural paradigm and the object-oriented paradigm. The paradigm should be chosen keeping in mind constraints, such as time, availability of resources, and nature of user's requirements.

• **Software design should demonstrate uniformity and integration:** In most cases, rules, format, and styles are defined in advance to the design team before the design work begins. The design is said to be integrated and uniform if the interfaces are properly defined among design components.

• **Software design should be structured to adapt change:** The fundamental design concepts (abstraction, refinement, modularity) should be applied to achieve this principle.

• **Software design should appraise to minimise conceptual (semantic) errors:** Design team must ensure that major conceptual elements of design, such as ambiguousness and inconsistency are addressed in advance before dealing with the syntactical errors present in the design model.

• **Software design should be structured to degrade gently:** Software should be designed to handle unusual changes and circumstances, and if need arise for termination; it must do so in a proper manner so that functionality of the software is not affected.

Software design should 'minimise the intellectual distance' between the software and problem existing in the real world: The design structure should be such that it always relates with the real-world problem.

• **Code reuse:** There is a common saying among software engineers: 'do not reinvent the wheel'. Therefore, existing design hierarchies should be effectively reused to increase productivity.

• **Designing for testability:** A common practice that has been followed is to separate testing from design and implementation. That is, the software is designed, implemented, and then handed over to the testers, who subsequently determine whether or not the software is fit for distribution and subsequent use by the customer. However, it has become apparent that the process of separating testing is seriously flawed, as discovering these types of errors after implementation usually requires the entire or a substantial part
of the software to be redone. Thus, the test engineers should be involved from the very beginning. For example, they should work with the requirements analysts to devise tests that will determine whether the software meets the requirements or not.

• **Prototyping:** Prototyping should be used to explore those aspects of the requirements, user interface, or software's internal design, which are not easily understandable. Using prototyping a quick 'mock-up' of the system can be developed. This mock up can be used as a highly effective means to highlight misconceptions and reveal hidden assumptions about the user interface and how the software should perform. Prototyping also reduces the risk of designing software that does not fulfil customer's requirements.

Note that design principles are often constrained by the existing hardware configuration, the implementation language, the existing file and data structures, and the existing organizational practices. Also, the evolution of each software design should be meticulously designed for future evaluations, references, and maintenance.

Developing a Design Model

To develop a complete specification of design (design model), four elements are used. These are:

• **Data design:** Creates data structure by converting data objects specified during analysis phase. The data objects, attributes, and relationships defined in entity relationship diagrams provide the basis for data design activity. Various studies suggest that design engineering should begin with data design, since this design lays the foundation for all other design elements.
• **Architectural design:** Specifies the relationship between structural elements of software, design patterns, architectural styles, and the factors affecting the way in which architecture can be implemented.

Data Design

Data design is the first of the design activities, which leads to better program structure, effective modularity, and reduced complexity. Data design is developed by transforming the data dictionary and entity relationship diagram (identified during the requirements phase) into data structures that are required to implement the software. The data design process includes identifying the data, defining specific data types and storage mechanisms, and ensuring data integrity by using business rules and other run-time enforcement mechanisms.
The selection process may involve algorithmic analysis of alternative structures in order to determine the most efficient design or the use of a set of modules that provides operations on some representation of objects.

Some principles are followed while specifying the data, which are listed below:
• All data structures and the operations to be performed should be identified.
• Data dictionary should be established and used.
• Low-level data design decisions should be deferred until late in the design process.
• The representation of data structure should be known to only those modules that directly use the data.
• A library of useful data structure and the operations that may be applied to them should be developed.
• Language should support abstract data types.
The structure of data can be viewed at three levels, namely, program component level, application level, and business level. At the **program component level**, the design of data structures and the algorithms required to manipulate them is necessary if high-quality software is desired. At the **application level**, the translation of a data model into a database is essential to achieve the specified business objectives of a system. At the **business level**, the collection of information stored in different databases should be reorganised into data warehouse, which enables data mining that has influential impact on the business.

Architectural Design

Requirements of the software should be transformed into an architecture that describes software's top-level structure and identifies its components. This is accomplished through architectural design (also called system design), which acts as a preliminary 'blueprint' from which software can be developed. IEEE defines architectural design as "*the process of defining a collection of hardware and software components and their interfaces to establish the framework for the development of a computer system*". This framework is established by examining the software requirement document and building a *physical model* using recognised software engineering methods.
The physical model describes the solution in concrete and implementation terms, which is used to produce a structured set of component specifications (each specification defines the functions, inputs and outputs of the component) that are consistent, coherent and complete. An architectural design performs the following functions:
• Provides a level of abstraction at which the software designers can specify the system behaviour (such as function and performance).
• Serves as the 'conscience' for a system as it evolves. By characterizing the crucial system design assumptions, a good architectural design guides the process of system enhancement indicating what aspects of the system can be easily changed without compromising system integrity.
• Evaluates all top-level designs.

• Develops and documents top-level design for the external and internal interfaces.
• Develops preliminary versions of user documentation.
• Defines and documents preliminary test requirements and the schedule for software integration.
Architectural design is derived from the following sources:
• Information regarding the application domain for the software to be developed.
• Using data flow diagrams.
• Availability of architectural patterns and architectural styles.
Architectural design occupies a pivotal position in software engineering. It is during architectural design that crucial requirements, such as performance, reliability, costs, and more are addressed. This task is cumbersome as the software engineering paradigm is shifting from monolithic, stand-alone, built-from-scratch systems to componentised, evolvable, standards-based, and product line oriented systems. Also, one key challenge for designers is to know precisely how to proceed from requirements to architectural design.
To avoid all these problems, designers adopt strategies such as reusability, componentisation, platform-based, standards-based, and so on.
While the architectural design is the responsibility of developers, participants in the architectural design phase should also include user representatives, systems engineers, hardware engineers, and operations personnel. In reviewing the architectural design, project management should ensure that all parties are consulted in order to minimise the risk of incompleteness and error.

Architectural Design Representation
Architectural design can be represented using various models, which are listed below:
• **Structural model:** Illustrates architecture as an ordered collection of programs components.
Framework model: Attempts to identify repeatable architectural design patterns, which are encountered in similar types of application. This leads to an increase in the level of abstraction.
• **Dynamic model:** Specifies the behavioural aspect of the software architecture and indicates how the structure or system configuration changes as the function changes due to change in external environment.
• **Process model:** Focuses on the design of the business or technical process, which must be implemented in the system.
• **Functional model:** Represents functional hierarchy of a system.
Architectural Design Output: The output of the architectural design process is an architectural design document (ADD), which consists of a number of graphical representations that consist of software models along with associated

descriptive text. These models include static model, an interface model, a relationship models, and a dynamic process model that shows how the system is organized into process at run-time. This document gives the developers' solution to the problem stated in the software requirements specification (SRS) but avoids the detailed consideration of software requirements that do not affect the structure. In addition to ADD, other outputs of the architectural design are:

• Progress reports, configuration status accounts and audit reports.
• Various plans for detailed design phase, which includes:
  - Software project management plan.
  - Software configuration management plan.
  - Software verification and validation plan.
  - Software quality assurance plan.

4.4.2 Architectural Styles

Architectural styles define a family of systems in terms of a pattern of structural organization.

It also characterises a family of systems that are related by sharing structural and semantic properties. In short, the objective of using architectural styles is to establish a structure for all the components present in a system. If an existing architecture is to be reengineered, then imposition of an architectural style results in fundamental changes in the structure of the system. This change also includes reassignment of the functionality performed by the components.

By constraining the design space, an architectural style often permits specialised and style specific analyses. Also, an architectural style makes it easier for other stakeholders to understand a system's organization if conventional structures are used.Computer-based system (software is part of this system) exhibits one of the many available architectural styles. Each style describes a system category that includes the following:

• *Computational components* (such as clients, server, filter, database) that perform a function required by the system.
• A set of *connectors* that enable interactions and co-ordination among these components (such as procedure call, events broadcast, database protocols, and pipes).
• *Constraints* that define integration of components to form a system.
• *Semantic model*, which enables software designer to understand the overall properties of a system by analysing the known properties of its constituent parts.

Functional Independence

Functional independence is the refined form of the design concepts of modularity, abstraction, and information hiding. Functional independence is achieved by developing a module in such a way that uniquely performs given sets of function without interacting with other parts of the system. Software that uses property of functional independence is easier to develop because its function can be categorised in a systematic manner. Moreover, independent modules require less maintenance and testing activity as secondary effect caused by design modification are limited with less propagation of errors. In short, it can be said that functional independence is a key to a good software design and a good design results in high quality software.

There exist two qualitative criteria for measuring functional independence, namely, coupling and cohesion. **Coupling** is a measure of relative interconnection among modules whereas **cohesion** measures the relative functional strength of a module.

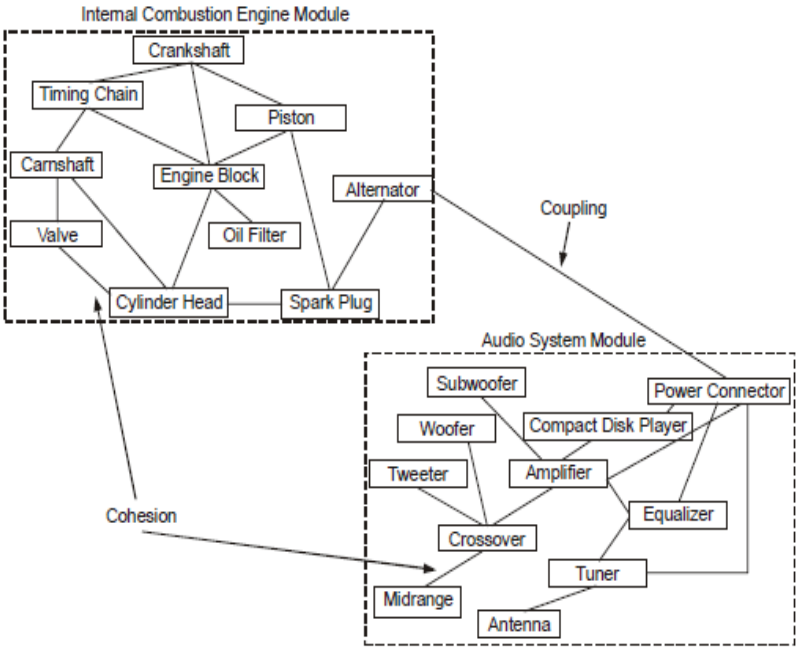

Figure 4.10 Coupling and Cohesion

(*a*) *Coupling*: Coupling is the measure of interdependence between one module and another. Coupling depends on the interface complexity between components, the point at which entry or reference is made to a module, and the kind of data that passes across an interface. For better interface and well-structured system, modules should have low coupling, which minimises the 'ripple effect' where changes in one module cause errors in other modules

(*b*) *Cohesion*: Cohesion is the measure of strength of the association of elements within a module. A cohesive module performs a single task within a software procedure, which has less interaction with procedures in other part of the program. In practice, designer should avoid low-level of cohesion when designing a module. Generally, low coupling results in high cohesion and vice versa.

User Interface Design

User interfaces determine the way in which users interact with the software. The user interface design creates effective communication medium between a human and a computing machine. It provides easy and intuitive access to information as well as efficient interaction and control of software functionality. For this, it is necessary for the designers to understand what the user needs and wants from the user interface.

User Interface Rules

Designing a good and efficient user interface is a common objective among software designers. But what makes a user interface looks 'good'? Software designers strive to achieve a good user interface by following three rules, namely, *ease of learning*, *efficiency of use*, and *aesthetic appeal*.

(*a*) *Ease of Learning*: Ease of learning describes how quickly and effortlessly users learn to use the software. Ease of learning is primarily important for new users. However, even experienced users face a learning experience problem when they attempt to expand their usage of the product or when they use a new version of the software. Here, the *principle of state visualisation* is applied, which states that each change in the behaviour of the software should be accompanied by a corresponding change in the appearance of the interface.

Developers of software applications with a very large feature set can expect only few users to have mastered the entire feature set. Thus, designers of these applications should be concerned about the ease of learning for otherwise experienced users.

Generally to ease the task of learning, designers make use of the tools listed below:

• **Affordance:** Provides clues that suggest what a machine or tool can do and how to use it. For example, the style of a door handle on the doors of many departmental stores, offices, and shops suggest whether to pull a door or push a door to open. If the wrong style door handle is used, people struggle with the door. In this way, the door handle is more than just a tool for physically helping you to open the door; it is also an affordance showing you how the door opens. Similarly, software designers while developing user interface should offer hints as to what each part does and how it works.

• **Consistency:** Designers strive to maintain consistency within the interface. Every aspect of the interface, including seemingly minor details, such as font usage and colours is kept consistent when the behaviour is consistent. Here, *principle of coherence* (that is behaviour of the program should be internally and externally consistent) is applied.

Internal consistency means that the program's behaviour must make "sense" with respect to other parts of the program. For example, if one attribute of an object (for example, colour) is modified using a pop-up menu, then it is expected that other attributes of the object will also be edited in a similar manner. **External consistency** means that the program is consistent with the environment in which it runs. This includes consistency with both the operating system and other suite of applications that run within that operating system.

(*b*) *Efficiency of Use*: Once a user knows how to perform tasks, the next question is how efficiently can the user solve problems with the software? Efficiency can be evaluated reasonably only if users are no longer engaged in learning how to do the task and are rather engaged in performing the task.

Defining an efficient interface requires a deep understanding of the behaviour of target audience. How frequently do they perform the task? How frequently do they use the interface devices? How much training do they have? How distracted are they? A few guidelines help in designing an efficient interface.

• The task should require minimal physical actions. The desire of experienced users for hot keys and to shortcuts to pull-down menu actions is a well-known example of reducing the number of actions required to perform a task.

• The task should require minimal mental effort as well. A user interface, which requires the user to remember specific details will be less popular than one that remembers those details for the user. Similarly, an interface, which requires the user to make many decisions, particularly non-trivial decisions, will be less popular than the one that requires the user to make fewer or simpler decisions.

(*c*) *Aesthetically Pleasing*: Today, look and feel is one of the biggest USP (unique selling point) while designing software. An attractive user interface

improves the sales because people like to have things that look nice. An attractive user interface makes the user feel better (as it provides ease of use), while using the product. Many software organisations focus specifically on designing software, which has attractive look and feel so that they can lure customers/users towards their product(s).

In addition to the above-mentioned goals, there exist a *principle of metaphor* which if applied to the software's design result in a better and effective way of creating a user interface. This principle states that a complex software system can be understood easily if the user interface is created in a way that resembles an already developed system. For example, the popular Windows operating system uses similar (not same) look and feel in all of its operating system so that the users are able to use it in a user-friendly manner.

User Interface Design Process

User interface design, like all other software design elements, is an iterative process. Each step in user interface design occurs a number of times, each elaborating and refining information developed in the previous step. Although many different user interface design models have been proposed, all have the following steps in common.

1. Using information stated in the requirements document, each task and actions are defined.
2. A complete list of events (user actions), which causes the state of the user interface to change is defined.
3. Each user action is assigned iteration.
4. Each user interface state, as it will appear (look) to the user is depicted.
5. Indicate how user interprets the state of the system using information provided through the interface.

To sum up, the user interface design activity starts with the identification of user, task, and environmental requirements. After this, user states are created and analysed to define a set of interface objects and actions. These object then form the basis for the creation of screen layout, menus, icons, and much more.

While designing the user interface, following points must be kept in mind:

• Follow the rules stated in section 4.5.1. Any interface that fails to achieve any of these rules to a reasonable degree needs to be re-designed.
• Determine how interface will be implemented.
• Consider the environment (like operating system, development tools, display properties, and so on).

Evaluating User Interface Design

User interface design process generates a useful interface, however, no designer should expect to achieve a high quality interface on the first go. Each iteration of the steps involved in user interface design process leads to development of a prototype. The objective of developing a prototype is to capture the '*essence*' of the user interface. This prototype is evaluated, discrepancies are detected and accordingly re-design takes place. This process carries on until a good interface evolves of software behaviours. Choosing an appropriate evaluation technique helps in knowing whether a prototype is able to achieve the desired user interface or not.

Evaluation Techniques: To provide a feedback for the next iteration, each interface must be evaluated in some manner. This evaluation must indicate what works well with the interface, and more importantly, what are the areas of improvement. Some of the evaluation techniques used to evaluate the user interface design are: *use it yourself*, *colleague evaluation*, *user testing*, and *heuristic evaluation*. Each technique offers unique advantages and limitations. To a varying degree, they highlight ease of learning or efficiency issues. The third goal of aesthetic pleasing largely depends on user to user and can be best evaluated by observing what people find pleasing.

• **Use it yourself:** In this evaluation technique working through a number of the tasks defined by the requirements often indicates a myriad of problems in the initial design(s).

This evaluation technique helps in identifying the entire missing pieces of the interface and significant inefficient usage issues.

• **Colleague evaluation:** Since the designers are aware of the functionality of the software, it is possible that they may miss out on issues of ease of learning and efficiency.

Showing the interface to a colleague may help in solving these issues. However, note that unless the prototype is inherently useful in its current state, colleagues are unlikely to use the prototype for sufficient time to discover many efficiency issues.

• **User testing:** This testing is considered to be the most realistic form of interface evaluation. Users can test the prototypes, with the expected differences recorded in a feedback. The most useful feedback is often the comments the users make to each other as they try to understand how to accomplish the task. Before performing user testing, the designer should choose one or more tasks that the user will be expected to perform. Any necessary background information must be prepared in advance, with the user having sufficient time to understand this background before beginning the test.

User testing is one of the best tools available for evaluating ease of learning. However, it offers little or no assistance in discovering efficiency issues.

• **Heuristic evaluation:** Such evaluations are inherently subjective, where each evaluator finds a different set of problems in the interface. Generally, experienced user interface designers are able to find more errors as compared to less experienced designers. Heuristic evaluation provides a checklist of issues, which should be considered for each iteration of user interface design. This checklist considers the following categories:

 Simple and natural dialogue: An interface provides a simple and natural dialogue if it gently leads the user from one action to the next for common tasks.

 Graphic design and colour: Consistency in the use of colour and other graphic design elements helps the user. The colour and font can be used to convey information subtly.

 Less is more: Keep the information being presented to minimum. A cluttered interface is both aesthetically displeasing and confusing to navigate.

 Speak the user's dialogue: Always use the user's terminology and phrase text from the user's perspective. For example, messages should display "you have purchased..." and not "we have sold you...". When you choose terminology and metaphors, be sure that they are consistent, both internally and with outside expectations.

 Minimize the user's memory load: Minimize the user's memory load by presenting as much information as appropriate, when required by the user.

 Be consistent: Consistency is one of the designer's most desired objectives. Be sure that similar visual descriptions for similar items are used.

Provide feedback: Always provide the user with clear feedback about what the program is doing.

 Provide shortcuts: Providing shortcuts, such as hot keys, are an important aspect of efficiency. These shortcuts enhance the work experience for experienced users.

 Provide error messages: Appropriate error messages let the user know exactly what is wrong and how to fix it. The message should indicate which part of the input is incorrect.