Unit-III Software Design

Introduction

A software design creates meaningful engineering representation (or model) of some software product that is to be built. Designers must strive to acquire a repertoire of alternative design information and learn to choose the elements that best match the analysis model. A design model can be traced to the customer's requirements and can be assessed for quality against predefined criteria. During the design process the software requirements model (data, function, behavior) is transformed into design models that describe the details of the data structures, system architecture, interfaces, and components necessary to implement the system. Each design product is reviewed for quality (i.e. identify and correct errors, inconsistencies, or omissions, whether better alternatives exist, and whether the design model can be implemented within the project constraints) before moving to the next phase of software development.

Principles of Software Design

- Encompasses the set of principles, concepts, and practices that lead to the development of a high quality system or product
- Design principles establish and overriding philosophy that guides the designer as the work is performed
- Design concepts must be understood before the mechanics of design practice are applied
- Goal of design engineering is to produce a model or representation that is bug free (firmness), suitable for its intended uses (commodity), and pleasurable to use (delight)
- Software design practices change continuously as new methods, better analysis, and broader understanding evolve

Software Engineering Design

- Data/Class design - created by transforming the analysis model class-based elements (class diagrams, analysis packages, CRC models, collaboration diagrams) into classes and data structures required to implement the software
- Architectural design - defines the relationships among the major structural (fundamental, essential,)elements of the software, it is derived from the class-based elements and flow-oriented elements (data flow diagrams, control flow diagrams, processing narratives) of the analysis model

- Interface design - describes how the software elements, hardware elements, and end-users communicate with one another, it is derived from the analysis model scenario-based elements (use-case text, use-case diagrams, activity diagrams, swim lane diagrams), flow-oriented elements, and behavioral elements (state diagrams, sequence diagrams)
- Component-level design - created by transforming the structural elements defined by the software architecture into a procedural (technical,practical) description of the software components using information obtained from the analysis model class-based elements, flow-oriented elements, and behavioral elements

Software Quality Attributes

A good design must
- implement all explicit requirements from the analysis model and accommodate all implicit requirements desired by the user
- be readable and understandable guide for those who generate code, test components, or support the system
- provide a complete picture (data, function, behavior) if the software from an implementation perspective

Design Quality Guidelines

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
  - can be implemented in an evolutionary manner
- be modular
- contain distinct representations of data, architecture, interfaces, and components (modules)
- lead to data structures that are appropriate for the objects to be implemented and be drawn from recognizable design patterns
- lead to components that exhibit independent functional characteristics
- lead to interfaces that reduce the complexity of connections between modules 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

FURPS Quality Factors

- Functionality – feature set and program capabilities
- Usability – human factors (aesthetics, consistency, documentation)
- Reliability – frequency and severity opf failure
- Performance – processing speed, response time, throughput, efficiency
- Supportability – maintainability (extensibility, adaptability, serviceability), testability, compatibility, configurability

Generic Design Task Set

1. Examine information domain model and design appropriate data structures for data objects and their attributes
2. Select an architectural pattern appropriate to the software based on the analysis model
3. Partition the analysis model into design subsystems and allocate these subsystems within the architecture
    - Be certain each subsystem is functionally cohesive
    - Design subsystem interfaces
    - Allocate analysis class or functions to subsystems
4. Create a set of design classes
    - Translate analysis class into design class
    - Check each class against design criteria and consider inheritance issues
    - Define methods and messages for each design class
    - Evaluate and select design patterns for each design class or subsystem after considering alternatives
    - Revise design classes and revise as needed
5. Design any interface required with external systems or devices
6. Design user interface
    - Review task analyses
    - Specify action sequences based on user scenarios
    - Define interface objects and control mechanisms
    - Review interface design and revise as needed
7. Conduct component level design
    - Specify algorithms at low level of detail
    - Refine interface of each component
    - Define component level data structures
    - Review components and correct all errors uncovered
8. Develop deployment model

Design Concepts

- Abstraction – allows designers to focus on solving a problem without being concerned about irrelevant lower level details (*procedural abstraction* - named sequence of events and *data abstraction* – named collection of data objects)
- Software Architecture – overall structure of the software components and the ways in which that structure provides conceptual integrity for a system
  - Structural models – architecture as organized collection of components
  - Framework models – attempt to identify repeatable architectural patterns
  - Dynamic models – indicate how program structure changes as a function of external events
  - Process models – focus on the design of the business or technical process that system must accommodate
  - Functional models – used to represent system functional hierarchy
- Design Patterns – description of a design structure that solves a particular design problem within a specific context and its impact when applied
- Separation of concerns – any complex problem is solvable by subdividing it into pieces that can be solved independently
- Modularity - the degree to which software can be understood by examining its components independently of one another
- Information Hiding – information (data and procedure) contained within a module is inaccessible to modules that have no need for such information
- Functional Independence – achieved by developing modules with single-minded purpose and an aversion to excessive interaction with other models
  - Cohesion - qualitative indication of the degree to which a module focuses on just one thing
  - Coupling - qualitative indication of the degree to which a module is connected to other modules and to the outside world
- Refinement – process of elaboration where the designer provides successively more detail for each design component
- Aspects – a representation of a cross-cutting concern that must be accommodated as refinement and modularization occur
- Refactoring – process of changing a software system in such a way internal structure is improved without altering the external behavior or code design

Design Classes

- Refine analysis classes by providing detail needed to implement the classes and implement a software infrastructure the support the business solution
- Five types of design classes can be developed to support the design architecture
  - user interface classes – abstractions needed for human-computer interaction (HCI)
  - business domain classes – refinements of earlier analysis classes
  - process classes – implement lower level business abstractions
  - persistent classes – data stores that persist beyond software execution
  - System classes – implement software management and control functions

Design Class Characteristics

- Complete (includes all necessary attributes and methods) and sufficient (contains only those methods needed to achieve class intent)
- Primitiveness – each class method focuses on providing one service
- High cohesion – small, focused, single-minded classes
- Low coupling – class collaboration kept to minimum

Design Model

- Process dimension – indicates design model evolution as design tasks are executed during software process
  - Architecture elements
  - Interface elements
  - Component-level elements
  - Deployment-level elements
- Abstraction dimension – represents level of detail as each analysis model element is transformed into a design equivalent and refined
  - High level (analysis model elements)
  - Low level (design model elements)
- Many UML diagrams used in the design model are refinements of diagrams created in the analysis model (more implementation specific detail is provided)
- Design patterns may be applied at any point in the design process

Data Design

- High level model depicting user's view of the data or information
- Design of data structures and operators is essential to creation of high-quality applications
- Translation of data model into database is critical to achieving system business objectives
- Reorganizing databases into a data warehouse enables data mining or knowledge discovery that can impact success of business itself

Architectural Design

- Provides an overall view of the software product
- Derived from
    - Information about the application domain relevant to software
    - Relationships and collaborations among analysis model elements
    - Availability of architectural patterns and styles
- Usually depicted as a set of interconnected systems that are often derived from the analysis packages with in the requirements model

Interface Design

- Interface is a set of operations that describes the externally observable behavior of a class and provides access to its public operations
- Important elements
    - User interface (UI)
    - External interfaces to other systems
    - Internal interfaces between various design components
- Modeled using UML communication diagrams (called collaboration diagrams in UML 1.x)

Component-Level Design

- Describes the internal detail of each software component
- Defines
    - Data structures for all local data objects
    - Algorithmic detail for all component processing functions
    - Interface that allows access to all component operations
- Modeled using UML component diagrams, UML activity diagrams, pseudocode (PDL), and sometimes flowcharts

Deployment-Level Design

- Indicates how software functionality and subsystems will be allocated within the physical computing environment
- Modeled using UML deployment diagrams
- *Descriptor form* deployment diagrams show the computing environment but does not indicate configuration details
- *Instance form* deployment diagrams identifying specific named hardware configurations are developed during the latter stages of design

Architectural Styles

An Architectural style typically specifies the design vocabulary ,constraints on how that vocabulary is used and semantic assumptions about that vocabulary. Each style has several views and structures. An architectural view represents a set of elements and the relationships among them. Thus an architectural style defines a family of such systems in terms of a pattern of structural organization.

Layered Architectural style:-

- This type of architectural style is the hierarchical organization of a system in layers.
- Layered systems are designed in a modular fashion at each layer in the architecture. There are well-defined interfaces between the layers.
- Layered system design is based on the increasing level of abstraction.
- Layered organization of an operating system is a good example of layered architectural style.
- Other examples can be a database system ,an object request broker ,network layers etc.
- This architecture promotes reuse.
- However it has some drawbacks. It is not necessary to design all systems in a layered fashion.
- Adding more number of layers may decrease system performance.

Data-Flow Style:-

- The data flow is characterized by viewing the system as a series of transformations in a successive manner.
- In this style the input data enters the system and then moves through the components one at a time. and finally the transformed data are produced as output.
- These styles focus on achieving the quality of reuse and modifiability.
- The pipe and filter style follows the component connector structure in which components are filters and pipes are connectors.
- The filter reads stream of data as input ,performs data transformation and forwards the output data stream to another filter.
- The pipe transforms data streams from one filter to another.
- The pipe and filter style processes data streams in a pipeline instead of processing them as a single entity in the batch sequential style.
- For example a compiler executes in a pipe and filter style. It is comprised of several pipelined activities or filters.
- These are lexical analysis ,semantic analysis, intermediate code generation and code generation.

Client-Server style:-

- It is useful for distributed processing, load balancing, separation of concerns and performance analysis.
- In this style there are two types of components, client and server.
- There exists connecting network between the clients and servers.
- The clients request services and the server provides services to the clients.
- The client communicates with servers through protocol and message connectors.

- For example a web browser program running on the internet has client-server style.

- This architectural style provides higher security ,centralized data access and easier maintenance.

Shared Data Style:-

- It is also called repository style which is comprised of a central data repository and a number of data accessors connected to the central repository.

- It is the shared data such as databases, files etc which are used by data accessors such as users, nodes, developers.

- Shared data style creates, stores, updates and accesses persistent data.

- Data accessors communicate with the help of read and write connectors.

- There are two major categories of shared data styles namely black board and traditional database

- In the black board style, any change or update in the repository will be of benefit to all the data accessors.

- For example, an insurance company policy has a central repository in which any policy change or update is informed to the customers.