Design Concepts and Principles: Overview

A software design is a meaningful engineering representation of some software product that is to be built. A design 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 is transformed into design models that describe the details of the data structures, system architecture, interface, and components. Each design product is reviewed for quality before moving to the next phase of software development.

Design Specification Models

- Data design - created by transforming the analysis information model (data dictionary and ERD) into data structures required to implement the software
- Architectural design - defines the relationships among the major structural elements of the software, it is derived from the system specification, the analysis model, and the subsystem interactions defined in the analysis model (DFD)
- Interface design - describes how the software elements communicate with each other, with other systems, and with human users; the data flow and control flow diagrams provide much the necessary information
- Component-level design - created by transforming the structural elements defined by the software architecture into procedural descriptions of software components using information obtained from the PSPEC, CSPEC, and STD

Design Guidelines

A design should

- exhibit good architectural structure
- 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 reputable method that is driven by information obtained during software requirements analysis

Design Principles

The design

- process should not suffer from tunnel vision
- should be traceable to the analysis model
- should not reinvent the wheel
- should minimize intellectual distance between the software and the problem as it exists in the real world
- should exhibit uniformity and integration
- should be structured to accommodate change
- should be structured to degrade gently, even with bad data, events, or operating conditions are encountered
- should be assessed for quality as it is being created
- should be reviewed to minimize conceptual (semantic) errors

Fundamental Software 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, data abstraction - named collection of data objects)
- Refinement - process of elaboration where the designer provides successively more detail for each design component
- Modularity - the degree to which software can be understood by examining its components independently of one another
- Software architecture - overall structure of the software components and the ways in which that structure provides conceptual integrity for a system
- Control hierarchy or program structure - represents the module organization and implies a control hierarchy, but does not represent the procedural aspects of the software (e.g. event sequences)
- Structural partitioning - horizontal partitioning defines three partitions (input, data transformations, and output); vertical partitioning (factoring) distributes control in a top-down manner (control decisions in top level modules and processing work in the lower level modules)
- Data structure - representation of the logical relationship among individual data elements (requires at least as much attention as algorithm design)
- Software procedure - precise specification of processing (event sequences, decision points, repetitive operations, data organization/structure)
- Information hiding - information (data and procedure) contained within a module is inaccessible to modules that have no need for such information

Modular Design Method Evaluation Criteria

- Modular decomposability - provides systematic means for breaking problem into subproblems
- Modular composability - supports reuse of existing modules in new systems
- Modular understandability - module can be understood as a stand-alone unit
- Modular continuity - side-effects due to module changes minimized
- Modular protection - side-effects due to processing errors minimized

Control Terminology

- Span of control (number of levels of control within a software product)
- Depth (distance between the top and bottom modules in program control structure)
- Fan-out or width (number of modules directly controlled by a particular module)
- Fan-in (number of modules that control a particular module)
- Visibility (set of program components that may be called or used as data by a given component)
- Connectivity (set of components that are called directly or are used as data by a given component)

Effective Modular Design

- Functional independence - modules have high cohesion and low coupling
- 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

Design Heuristics for Effective Modularity

- Evaluate the first iteration of the program structure to reduce coupling and improve cohesion.
- Attempt to minimize structures with high fan-out; strive for fan-in as structure depth increases.
- Keep the scope of effect of a module within the scope of control for that module.

- Evaluate module interfaces to reduce complexity, reduce redundancy, and improve consistency.
- Define modules whose function is predictable and not overly restrictive (e.g. a module that only implements a single subfunction).
- Strive for controlled entry modules, avoid pathological connection (e.g. branches into the middle of another module)

Architectural design: Overview

Architectural design represents the structure of the data and program components required to build a computer-based system. A number of architectural "styles" exist. Architectural design begins with data design and proceeds to the derivation of one or more representations of the architectural structure of the system. The resulting architectural model encompasses both the data architecture and the program structure. The architectural model is subjected to software quality review like all other design work products.

Software architecture
Software architecture is a representation that enables a software engineer to

- Analyze the effectiveness of the design in meeting stated requirements
- Consider architectural alternatives
- Reduce the risk associated with the construction of the software
- Examine the system as a whole

Data Design Principles

- Systematic analysis principles applied to function and behavior should also be applied to data.
- All data structures and the operations to be performed on each should be identified.
- Data dictionary should be established and used to define both data and program design.
- Low level design processes should be deferred until late in the design process.
- Representations of data structure should be known only to those modules that must make direct use of the data contained within in the data structure.
- A library of useful data structures and operations should be developed.
- A software design and its implementation language should support the specification and realization of abstract data types.

Architectural Styles

- Data centered - data store (e.g. file or database) lies at the center of this architecture and is accessed frequently by other components that modify data
- Data flow - input data is transformed by a series of computational or manipulative components into output data
- Call and return - program structure decomposes function into control hierarchy with main program invokes several subprograms
- Object-oriented - components of system encapsulate data and operations, communication between components is by message passing
- Layered - several layers are defined, each accomplishing operations that progressively become closer to the machine instruction set

Architecture Design Assessment Questions

- How is control managed within the architecture?
- Does a distinct control hierarchy exist?
- How do components transfer control within the system?
- How is control shared among components?

- What is the control topology?
- Is control synchronized or asynchronous?
- How are data communicated between components?
- Is the flow of data continuous or sporadic?
- What is the mode of data transfer?
- Do data components exist? If so what is their role?
- How do functional components interact with data components?
- Are data components active or passive?
- How do data and control interact within the system?

Architecture Trade-off Analysis Method

1. Collect scenarios
2. Elicit requirements, constraints, and environmental description
3. Describe architectural styles/patterns chosen to address scenarios and requirements (module view, process view, data flow view)
4. Evaluate quality attributes independently (e.g. reliability, performance, security, maintainability, flexibility, testability, portability, reusability, interoperability)
5. Identify sensitivity points for architecture (any attributes significantly affected by variation in the architecture)
6. Critique candidate architectures (from step 3) using the sensitivity analysis (conducted in step 5)

Architectural Complexity (similar to coupling)

- Sharing dependencies - represent dependence relationships among consumers who use the same resource or producers who produce for the same consumers
- Flow dependencies - represent dependence relationships between producers and consumers of resources
- Constrained dependencies - represent constraints on the relative flow among a set of components

Mapping Requirements to Software Architecture in Structured Design

- Establish type of information flow (transform flow - overall data flow is sequential and flows along a small number of straight line paths; transaction flow - a single data item triggers information flow along one of many paths)
- Flow boundaries indicated
- DFD is mapped into program structure
- Control hierarchy defined
- Resultant structure refined using design measures and heuristics
- Architectural description refined and elaborated

Transform Mapping

- Review fundamental system model
- Review and refine data flow diagrams for the software
- Determine whether the DFD has transform or transaction characteristics
- Isolate the transform center by specifying incoming and outgoing flow boundaries
- Perform first level factoring
- Perform second level factoring
- Refine the first iteration architecture using design heuristics for improved software quality

Transaction Mapping

- Review fundamental system model
- Review and refine data flow diagrams for the software

- Determine whether the DFD has transform or transaction characteristics
- Identify the transaction center and flow characteristics along each action path
- Map the DFD to a program structure amenable to transaction processing
- Factor and refine the transaction structure and the structure of each action path
- Refine the first iteration architecture using design heuristics for improved software quality

Refining Architectural Design

- Processing narrative developed for each module
- Interface description provided for each module
- Local and global data structures are defined
- Design restrictions/limitations noted
- Design reviews conducted
- Refinement considered if required and justified

Component level design: Overview

The purpose of component level design is to translate the design model into operational software. Component level design occurs after the data, architectural, and interface designs are established. Component-level design represents the software in a way that allows the designer to review it for correctness and consistency, before it is built. The work product produced is the procedural design for each software component, represented using graphical, tabular, or text-based notation.

Structured Programming

- Each block of code has a single entry at the top
- Each block of code has a single exit at the bottom
- Only three control structures are required: sequence, condition (if-then-else), and repetition (looping)
- Reduces program complexity by enhancing readability, testability, and maintainability

Design Notation

- Flowcharts (arrows for flow of control, diamonds for decisions, rectangles for processes)
- Box diagrams (also known as Nassi-Scheidnerman charts - process boxes subdivided to show conditional and repetitive steps)
- Decision table (subsets of system conditions and actions are associated with each other to define the rules for processing inputs and events)
- Program Design Language (PDL - structured English or pseudocode used to describe processing details)

Program Design Language Characteristics

- Fixed syntax with keywords providing for representation of all structured constructs, data declarations, and module definitions
- Free syntax of natural language for describing processing features
- Data declaration facilities for simple and complex data structures
- Subprogram definition and invocation facilities

Design Notation Assessment Criteria

- Modularity (notation supports development of modular software)
- Overall simplicity (easy to learn, easy to use, easy to write)

- Ease of editing (easy to modify design representation when changes are necessary)
- Machine readability (notation can be input directly into a computer-based development system)
- Maintainability (maintenance of the configuration usually involves maintenance of the procedural design representation)
- Structure enforcement (enforces the use of structured programming constructs)
- Automatic processing (allows the designer to verify the correctness and quality of the design)
- Data representation (ability to represent local and global data directly)
- Logic verification (automatic logic verification improves testing adequacy)
- Easily converted to program source code (makes code generation quicker)

REFER TO CHAPTER 13, 14, and 16 ***"Pressman. R. S., Software Engineering a practitioners Approach. 5th Edition"*** ACCORDINGLY.