Design Fundamentals
◆ creative process
◆ set of guidelines and preferably a method is necessary to provide structure to the task in large systems
◆ process of determining how a system can be built so that it will behave in the manner described in the developer centered requirements specification

Purpose of Design
specify a solution to a given problem which is essentially expressed as a functional specification.
designer postulates a solution, models it, evaluates it against the original requirements and after some iteration produces the detailed specification of the solution for the “programmer” to implement

Objectives of Design
◆ provide blueprints (specification)
  – static structure of the solution
  – the way the pieces fit together
  – data objects to be used
  – algorithms to be used
  – packaging of the system

Problem Solving Process
◆ problem
  – stated in requirements specification
◆ solution
  – evaluation different options
  – making choices using design criteria
  – series of trade-offs
    » size, speed, ease of adaptation, etc.

Ultimate criteria
◆ Fitness of Purpose
  – solution should exhibit the best possible structure, but also do the required job as well as possible within the constraints
◆ Robustness
  – ability to withstand change
**Wicked Problem**
the solution to one of its aspects may reveal an even more serious difficulty

**Design Phases**
◆ architectural design
  – concerned with the general structure of the solution
  – define the relationships among the major structural elements
◆ interface design
  – defines how the software communicates with itself, with the systems it interacts, and with the humans who use it
◆ detailed design
  – formulation of blueprints for the particular solution
  – refine algorithms
  – determine data implementation

**Design Method**
◆ provides assistance with model building and with the translation process
◆ plan of action based on a set of decision making criteria
◆ supported by diagramming and symbolic forms

**Recording the Design Process**
◆ record the history of the evolution of a design
◆ in particular the reasons for making specific choices
  – helps with design audits and maintenance
    » maintenance needs to know why some choices were made and others discarded
◆ included in the design document

**Relationship to Other Activities**
◆ based on the requirements
◆ cost of maintenance is related to design

**Principles of “Good” Design**
◆ Fitness of Purpose
  – be the best solution which works within the constraints
◆ Robustness
  – able to withstand change
  – Lehman’s 5 Laws of System Evolution
    » a system has a life of its own
    Law of continuing change
– a system that is used undergoes continuing change until it becomes more
cost-effective to replace it
Law of increasing complexity
– the disorder of a system increases unless specific work is done to maintain
or reduce the “disorder”

Desirable Design Features
◆ Simplicity
  – simple as possible but no simpler
◆ Separation of Concerns
  – different concepts and components should be separated out
◆ Information Hiding
  – information about the detailed form of such objects as data structure or
    interfaces should be kept local to a module and should not be “visible”
    outside that unit

Characteristics of “Good” Design
◆ implements all the explicit requirements and accommodates implicit
  requirements
◆ readable and understandable to coders, testers and maintainers
◆ offers a complete picture addressing data, functional and behavioral issues

Evaluate the Quality of a Design
◆ hierarchical
◆ modular -- independent
◆ both data and procedural abstractions
◆ driven by requirements
◆ able to withstand change -- robust

Quality Features
◆ External Factors
  – speed, reliability, correctness, usability
  – those things readily observed by users
◆ Internal Factors
  – Fitness of Purpose
  – Robustness
Classification of Systems

◆ Batch
  – all operating characteristics are essentially determined when it begins
    processing one or more data streams. Any changes that occur arise because
    of the contents of the streams.

◆ Reactive
  – event driven, almost always asynchronous and non-deterministic

◆ Concurrent
  – multiple threads of execution utilizing one or more processors.
  – must consider scheduling, synchronization and mutual exclusion

Not mutually exclusive, many systems are a combination of these

Design Principles

◆ Abstraction
  – consideration of a quality apart from a particular instance
  – IEEE- A view of the problem that extracts the essential information relevant
    to a particular problem and ignores the remainder of the information
  – allows for the architectural and detailed design
  – levels of generalization
  – procedural -- sequence of instructions that have a specific and limited
    function referred to as one word
  – data -- named collection of data that describes an object and the actions
    performed on that object (encapsulation)
  – control -- implies a program control mechanism w/out specifying exact
    mechanism for control

◆ Refinement -- Wirth 1971
  – decompose the problem into more detailed instructions, continue successively
    until all instructions are expressed in terms of a programming language

◆ Modularity
  – partitioning of a system / decomposition

Modularity

◆ IEEE
  – The extent to which the software is composed of discrete components such
    that a change to one component has minimal impact on another.

◆ Pressman
  – Software is divided into separately named and addressable components called
    modules that are integrated to satisfy problem requirements
Types of Modules

◆ Sequential
  – referred to and executed without interruption

◆ Incremental
  – can be interrupted prior to completion and restarted at the point of interruption

◆ Parallel
  – executes simultaneously with another module in concurrent multiprocessor environments

Characteristics of Modular Programs

Modules
◆ implement a single independent function
◆ perform a single logical task
◆ single entry and single exit point
◆ separately testable

Modular Programs
◆ entirely constructed of modules
◆ can be developed by teams

Some embedded and real-time systems cannot afford the memory and time overhead of modules, still use modular design and implement in-line

Software Architecture

◆ overall structure of the software
◆ hierarchical structure
◆ interaction between modules
◆ structure of data

Structure Chart

- specialized decomposition diagram
- shows module invocation
  o not seq., repetition or selection
- aids
  o integration testing/planning
  o maintenance
- rules
  o one box at top – root
  o control passed down level by level and is always passed back up to the invoking module
Control Hierarchy
◆ fan-in -- how many modules directly control a module
◆ fan-out – how many modules does the module control
◆ superordinate -- controlling module
◆ subordinate -- controlled module
◆ visibility -- set of components that may be invoked or used as data by a component, even if indirectly
◆ connectivity -- set of components that are directly invoked or used as data by a given module

Information Hiding
◆ Parnas 1971
◆ directed towards maintenance
  – no propagation of errors
◆ aids in parallel development
◆ modules are designed so that implementation details are hidden from other modules
  – localizes change
◆ public and private

Measures for assessing modules
◆ cohesion
  – IEEE -- the degree to which the tasks performed by a module are functionally related
  – relationships among the elements making up a module
  – interaction within a module
◆ coupling
  – IEEE -- a measure of the interdependence among modules
  – interaction between two modules

Goal in Designing Modules
Striving for modules which are highly cohesive (perform 1 task) and lowly coupled (independent)
Yielding that any unit can be replaced by an equivalent unit with little or no changes to other units (Robust)

Cohesion Levels
◆ Coincident
Logical Cohesion
◆ performs a series of related actions, one of which is selected by the calling module
◆ usually has many parameters, not all are used in all cases
◆ interface is difficult to understand
◆ severe maintenance problems
◆ difficult to reuse

Temporal Cohesion
◆ performs a series of actions related in time
◆ typically each piece is more strongly related to other modules
◆ all elements are executed at once
◆ leads to maintenance problems
◆ poor reusability

Procedural Cohesion
◆ performs a series of actions related by the sequence of steps to be followed by the product
◆ output of one action is the input to the next
◆ not reusable
**Communication Cohesion**
- effectively procedural cohesion except that it acts upon the same set of data
  ◆ better than procedural because the actions are more closely related but still not reusable

**Informational Cohesion**
◆ performs a number of actions each with its own entry point, with independent code for each, all performed on the same data structure
◆ each has one entry, one exit
◆ basis of oo
◆ different from logical as each actions code is independent whereas in logical its intertwined

**Functional Cohesion**
◆ performs exactly one action or achieves a single goal
◆ can be reused
◆ easily maintained and enhanced
◆ leads to fault isolation
◆ easier to understand

**Cohesion**
◆ a large program usually contains modules of different levels of cohesion
◆ in assigning levels if two or more are applicable always assign the lowest

**Coupling**
◆ Content   Worst/High
◆ Common   ↓
◆ Control
◆ Stamp   ↓
◆ Data     Best/Low

**Content Coupling**
◆ one module directly references the content of another
◆ examples
  - module p modifies state module q
  - module p uses local variable of q through some math displacement
  - module p branches into q
◆ a change requires a changes in the both
◆ can’t be reused without each other
**Common Coupling**
- both modules have access to the same global variables, same database, or read and write the same record
- common block in FORTRAN
- code is unreadable, maintenance problems
- change often results in side effects
- exposed to more data than it needs
- difficult to reuse

**Control Coupling**
- one passes an element of control to another
- one explicitly controls the logic or sequencing of the other (logical cohesion)
- the two modules are not independent so poor reuse possibility

**Stamp Coupling**
- pass data structure (array, record or pointer to structure) but the module only uses some of its elements
- not reusable
- uncontrolled data access

**Data Coupling**
- all arguments are homogenous data items
- either simple variable or a completely used data structure
- desirable goal
- maintenance easier
- less possibility of regression fault

**Design Heuristics**
- evaluate 1st iteration to reduce coupling and improve cohesion
  - explode -- increases cohesion (2 from 1)
  - implode -- reduces coupling (1 from 2)
- attempt to minimize structures with high fan-out, strive for fan-in as depth increases
- Keep scope of effect of a module within its scope of control
  - control -- subordinates and their subordinates
- Evaluate module interfaces to reduce complexity and redundancy and improve consistency
- Define modules predictable in function but avoid being over restrictive
– internal memory leads to unpredictable
– restrictions of size of local data overly restrictive and will require
  maintenance
◆ Controlled entry and exit -- single entry and exit
◆ Package software based on design constraints and portability requirements
  – packaging -- assembly of software for particular environment
    » overlay comes into play

Quality Issues
◆ Fitness of Purpose
  – does it meet the specifications
  – completeness
◆ Robust
  – designed for maintenance
◆ Assess Quality
  – reviews and expert opinions
  – performance

Design Methods
◆ no one method works for all classifications of systems
◆ reason use a method
  – systematic means of organizing and structuring the design process
  – set of criteria for making choices
  – significant for large systems gives a common set of design goals for all
    participants
◆ benefits maintenance
  – easier to model and assess the likely effects of changes since the maintenance
    designers can understand how original designers did things and build the
    same model
  – benefits organization of the project but does NOT provide a recipe for
    producing a design

Common Characteristics of Design Methods
◆ Translate analysis into design
◆ Notation for representation
  – graphical
◆ Heuristics for refining and partitioning
◆ Guidelines for QA
Design Strategies

Top-Down Decomposition
- divide and conquer
- functional decomposition
- full understanding of the problem at the outset is essential, since most of the important decisions must be made early in the design process
- different choices made early may result in significantly different solutions
  - must document choices
- clearly state the intended function
- divide, connect and check the intended function by expressing it as an equivalent structure of properly connected sub-functions, each solving part of the problem
- divide, connect and check each sub-function far enough to be comfortable

Problems
- When to stop dividing
  » no firm guidelines
- problem of replication and reconciliation
  » when more than one designer hard to recognize duplicate modules

Use as a preliminary step
- separate out concurrent components
- determine main modules
- dividing tasks among team members

Composition
- uses entities and objects and models these with the operations performed on them
- gradually build the solution by adding features and viewpoints

Design Methodologies
- Decomposition
  - based on data flow
    » Structured System Analysis and Structured Design (SSA/SD)
    » Structured Analysis and Design (SADT)
  - based on data structure
    » Jackson Method (JSP/JSD)
    » Warnier-Orr (LCP/LCS)
- Composition
  - object oriented (OOA/OOD)

Representation of Design
- typically graphical or tabular
◆ model the problem
  – data flow diagram, control-flow diagram, entity-relationship diagram, use-case diagram
◆ describe the structure
  – structure chart, class diagram
◆ describe the logical solution
  – decision tables & trees, state transition diagrams & matrix, finite state machines, PDL

Design Document
◆ design views
  – decomposition description
    » partitions the system into design entities
  – dependency description
    » relationships between entities and system resources
  – interface description
    » everything a designer, programmer, or tester needs to know to use the design entities
  – detail description
    » internal design details of the entity
◆ can be broken into architectural and detailed design documents
◆ use corresponding models of the design method
◆ may be a section for each major component of the software
◆ may correspond to different user classes
  – interface section for programmers and testers
  – quality evaluation section for QA personnel
  – identification section for cm personnel
  – detailed design section for programmers

Design Entity
◆ identification -- unique name, used for referencing and tracking
◆ type -- description of the kind of entity: subprogram, module, data store etc.
◆ purpose -- why entity exists, rationale for creating the entity, include any special requirements
◆ function -- what the entity does
◆ subordinates -- identification of all entities composing this entity
◆ dependencies -- relationship to other entities
◆ interface-- how to interact with this entity
◆ resources -- elements used by this entity which are external to it
◆ processing -- rules by which the entity achieves its function
◆ data -- description of the data elements which are internal to the entity