Software Design

software Design- “a system decomposition into modules—description of what each module is intended to do and of the relationship among the modules.”

- Design for Change

- Program Families
Design Process

Management Aspects:

1. Preliminary design (high level design)
2. Detail design (low level design)

Technical Aspects:

1. Data design
2. Architectural design
3. Procedure design
4. Interface design
Data Design

1. The systematic analysis principles applied to function and behavior should also be applied to data.

2. All data structures and the operations to be performed on each should be identified.

3. A data dictionary should be established and used to define both data and program design.

4. Low-level data design decisions should be deferred until late in the design process.

5. The representation of data structure should be known only to those modules that must make direct use of the data contained within the structure.

6. A library of useful data structures and the operations that may be applied to them should be developed.
To develop a modular program structure and represent the control relationships between modules.
Design Fundamentals

1. Abstraction
2. Refinement
3. Modularity
4. Software Architecture
5. Control Hierarchy
6. Data Structure
7. Software Procedure
8. Information Hiding
Functional Independence

coupling: is a measure of the relative interdependence among modules.

cohesion: is a measure of the relative functional strength of a module.
Coupling

- Coupling is a measure of interconnection among modules in a software structure.

- Coupling depends on the interface complexity between modules.
Coupling

**No direct coupling:** two modules are subordinate to different modules.

**Data coupling:** data passed via a module interface.

**Stamp Coupling:** data structure passed via a module interface.

**Control coupling:** control is passed via a "flag" on which decisions are made in a subordinate or supordinate module.

**External coupling:** Modules are tied to an environment external to software.
Common coupling: a number of modules reference a global data area.

Content coupling: one module makes use of data or control information ‘maintained within the boundary of another module.'
Cohesion

**Coincidental cohesion:** A module that performs a set of tasks that relate to each other loosely.

**Logical cohesion:** A module that performs tasks that are related logically.

**Temporal cohesion:** A module that performs tasks that are related by the fact that all must be executed with the same span of time.

**Procedure cohesion:** A module that processes elements that are related and must be executed in a specific order.
**Communicational cohesion:** A module that processes elements that all concentrate on one area of a data structure.

**Sequential cohesion:** The output data from an element is the input for the next element.

**Functional cohesion:** All of the elements are related to the performance of a single function.
Design Heuristics

- Evaluate the “First-Cut” Program Structure to Reduce Coupling and Improve Cohesion.

- Attempt to Minimize Structures with High Fan-Out; Strive for Fan-In as Depth Increases.

- Keep Scope of Effect of a Module within the scope of Control of That Module.

- Evaluate Module Interfaces to Reduce Complexity and Redundancy and Improve Consistency.
• Define Modules Whose Function is Predictable, But Avoid Modules That Are Overly Restrictive.

• Strive for Single-Entry-Single-Exit Modules, Avoiding “Pathological Connections.”

• Package Software Based on Design Constraints and Portability Requirements.
Object is a component of the real world that is mapped into the software domain.

Encapsulation is an act of grouping into a single object both data and the operations that affect data.

Information-hiding: The private side of an object is how it does these things, and it can do them in any way required. How it performs the operations or computes the information is not a concern of other parts of the system.
Classes, Instances, and Inheritance

- A class is a set of objects that each has the same characteristics.

- An individual object is an instance of a large class.

- All objects are members of a large class and inherit the private data structure and operations that have been defined for that class.
Classes

**Subclass** is a class that inherits behavior from another class. A subclass usually adds its own behavior to define its own unique kind of object.

**Superclass** is a class from which a specific behavior is inherited.

**Abstract Class:** class that is not intended to produce instances of itself.
- A *message* consists of the name of an operation and any required argument.

- *Polymorphism* is the ability of two or more classes of object to respond to the same message, each in its own way.
The Process of OOA/D

1. Find the classes in your system.

2. Determine what operations each class is responsible for performing, and what knowledge it should maintain.

3. Determine the ways in which objects collaborate with other objects in order to discharge their responsibilities.
The Process of OOA/D

- a list of classes within your application,

- a description of the knowledge and operations for which each class is responsible, and

- a description of collaborations between classes.
The Process of Object-Oriented Design - Exploratory Phase

1. Extract noun phrases from the specification and build a list.

2. Walk through various scenarios to explore possibilities. Record the results on design cards.
Classes

1. Extract noun phrases from the specification and build a list.

2. Look for nouns that may be hidden, and add them to the list.

3. Identify candidate classes from the noun phrases by applying the following guidelines:
   - Model physical objects.
   - Model conceptual entities.
   - Use a single term for each concept.
   - Be wary of the use of adjectives.
   - Model categories of objects.
   - Model external interfaces.
   - Model the values of an object’s attributes.
4. Identify candidates for abstract superclasses by grouping classes that share common attributes.

5. Use categories to look for classes that may be missing.

6. Write a short statement of the purpose of the class.
Responsibilities

1. Find responsibilities using the following guidelines:
   
   - Recall the purpose of each class, as implied by its name and specified in the statement of purpose.
   
   - Extract responsibilities from the specification by looking for actions and information.
   
   - Identify responsibilities implied by the relationships between classes.

2. Assign responsibilities to classes using the following guidelines:
   
   - Evenly distribute system intelligence.
   
   - State responsibilities as generally as possible.
   
   - Keep behavior with related information.
   
   - Keep information about one thing in one place.
• Share responsibilities among related classes.

3. Find additional responsibilities by looking for relationships between classes.

• Use “is-kind-of” relationships to find inheritance relationships.

• Use “is-analogous-to” relationships to find missing superclasses.

• Use “is-part-of” relationships to find other missing classes.
Collaborations

1. Find and list collaborations by examining the responsibilities associated with classes. Ask:
   
   - With whom does this class need to collaborate to fulfill its responsibilities?
   
   - Who needs to make use of the responsibilities defined for this class?

2. Identify additional collaborations by looking for these relationships between classes:
   
   - the “is-part-of” relationship,
   
   - the “has-knowledge-of” relationship, and
   
   - the “depend-upon” relationship.

3. Discard classes if no classes collaborate with them, and they collaborate with no other classes.
Hierarchies

1. Build hierarchy graphs that illustrate the inheritance relationships between classes.

2. Identify which classes are abstract and which are concrete.

3. Draw Venn diagrams representing the responsibilities shared between classes.

4. Construct class hierarchies using the following guidelines.
   - Model a ”kind-of” hierarchy.
   - Factor common responsibilities as high as possible.
   - Make sure that abstract classes do not inherit from concrete concrete classes.
• Eliminate classes that do not add functionality.

5. Construct the contracts defined by each class using the following guidelines:

• Group responsibilities that are used by the same clients.

• Maximize the cohesiveness of classes.

• Minimize the number of contracts per class.
Subsystems

1. Draw a complete collaborations graph of your system.

2. Identify possible subsystems within your design. Look for frequent and complex collaborations. Name the subsystems.
   - Classes in a subsystem should collaborate to support a small and strongly cohesive set of responsibilities.
   - Classes within a subsystem should be strongly interdependent.

3. Simplify the collaborations between and within subsystems.
   - Minimize the number of collaborations a class has with other classes or subsystems.
   - Minimize the number of classes and subsystems to which a subsystem delegates.
• Minimize the number of different contracts supported by a class or a subsystem.
1. Construct the protocols for each class. Refine responsibilities into sets of signatures that maximize the usefulness of classes.

   - Use a single name for each conceptual operation, wherever it is found in the system.

   - Associate a single conceptual operation with each method name.

   - If classes fulfill the same specific responsibility, make this explicit in the inheritance hierarchy.

   - Make signatures generally useful.

   - Provide default values for as many parameters as reasonable.

2. Write a design specification for each class.
3. Write a design specification for each subsystem.

4. Write a design specification for each contract.