1980s:

OO design: added inheritance, multiple inheritance, and polymorphism to ADT.

In process added complexity and increased some types of connectivity.

Lots of claimed advantages -- so far empirical evaluation is not supporting them well.

1990s:

Architecture

Patterns

Frameworks

Kits

etc.

Software Design Principles

- Design is a creative, problem-solving activity.
  - No recipe for doing it - always need some type of "magic".
  - Quality and expertise of designers is determinant for success.

    Simon: An expert has over 50,000 chunks of domain knowledge at hand.

      Solving a problem involves mapping into knowledge available.

      The larger this knowledge and the more accessible, the more successful the process will be.

Software Design Principles (2)

- Brooks, Curtis: Successful software development often depends on small number of exceptional designers who "think on a system level."
  - Curtis: Such people might not be particularly good programmers.

- Design problem: How to decompose system into parts
  - each with a lower complexity than system as a whole
  - while minimizing interaction between the parts

  such that the parts together solve the problem.

No universal way of doing this.

Four Primary Design Principles

1. Separation of concerns
   - Deal with separate aspects of a problem separate.
2. Abstraction
   - Identify important aspects of a phenomenon and ignore details that are irrelevant at this stage.
   - Hierarchical abstraction: build hierarchical layers of abstraction
     - Procedural (functional) abstraction
     - Data abstraction
     - Control abstraction (abstract from precise sequence of events handled, e.g., nondeterminacy)

Four Primary Design Principles (2)

3. Simplicity

   – Emphasis on software that is clear, simple, and therefore easy to check, understand, and modify.

4. Restricted visibility

   – Locality of information

General Software Design Concepts

Implementations of the general principles

- Decomposition
  - Can decompose with respect to time order, data flow, logical groupings, access to a common resource, control flow, or some other criterion.
  - Functional decomposition seems to be a natural way for people to solve problems as evidenced by its wide use.
  - Top-down decomposition: start at high levels of abstraction and progress to levels of greater and greater detail.
  - Bottom-up: form and layer groups of instruction sequences until work way up to a complete solution.

General Software Design Concepts (2)

- Decomposition (con't.)
  - Iterative decision making process:
    - List difficult decisions and decisions likely to change
    - Design a module specification to hide each such decision
    - Break module into further design decisions.
    - Continue refining until all design decisions hidden in a module
- Program Families: design for flexibility, not generality

General Software Design Concepts (3)

- Virtual Machines
  - A module provides a virtual machine: a set of operations that can be invoked in a variety of ways and orders to accomplish a variety of tasks.
  - Don’t think of systems in terms of components that correspond to steps in processing.
  - Do provide a set of virtual machines that are useful for writing many programs.
- Information Hiding
  - Each design unit hides internal details of processing activities.
  - Design units communicate only through well-defined interfaces.
  - Each design unit specified by as little information as possible
  - If internal details change, client units should need no change

General Software Design Concepts (4)

- Modularity
  - Separation of concerns:
    1. Deal with details of each module in isolation (ignoring details of other modules)
    2. Deal with overall characteristics of all modules and their relationships in order to integrate them into a coherent system.
  - Base on hierarchy and abstraction:
    - Abstraction handled through information hiding
    - Hierarchy by defining uses and is-composed-of relations
  - Minimize connectivity

General Software Design Concepts (5)

- Modularity (con’t.)
  - Sample things to modularize and encapsulate:
    - abstract data types
    - algorithms (e.g., sort)
    - input and output formats
    - processing sequence
    - machine dependencies (e.g., character codes)
    - policies (e.g., when and how to do garbage collection)
    - external interfaces (hardware and software)
  - Benefits:
    - Allows understanding each part of a system separately
    - Aids in modifying system
    - May confine search for a malfunction to a single module.

Design Methods

- Set of guidelines, heuristics, and procedures on how to go about designing a system.
- Usually offer a notation to express result of design process.
- Trying to provide a systematic means for organizing the design process and its products.
- Design method may be based on:
  - Functional decomposition
  - Data flow
  - Data structures
  - Control flow
  - Objects
- Vary in degree of prescriptiveness

David Budgen, Software Design Methods: Life Belt or Leg Iron (IEEE Software, Sept/Oct. 99)

Will the adoption of a design method help the software development process (be a "life belt") or is there significant risk that its use will lead to suboptimum solution (be a "leg iron")?

Argument:

- Two general design characteristics:
  1. "Wicked" nature of any design process:

     Adopting a particular solution approach to a problem may make task of solving it more intractible, i.e., the design process is not neutral.

  2. Expert designers engage in opportunistic behavior:

     As solution's form emerges, problem solving strategy is adapted to meet new characteristics that are revealed, i.e., expert designers do not follow a single method.

- These challenge the belief that good software engineering design solutions will most likely come from systematically following a prescriptive procedural method.

- 60s and 70s: people recognized that a systematic approach to development needed to cope with large-scale projects. Needed a way to promulgate and encourage the adoption of desirable practices.

  A procedural form (do this, then do this, then this ...) lent itself to this role.

  Also easily conveyed through books and courses, easy to teach, easy to write exam questions.

  Yourdan, Michael Jackson, etc.

  Met some real needs.

- By late 70s, use of procedural form was entrenched.

- But some good practices that did not lend themselves to such a form, e.g., information hiding (for which no satisfactory form of procedural development practice has yet been devised).

- Reaction in 80s to shortcomings was to "pile more on"
  - More diagrammatical forms
  - More models
  - More complexity

    "Arguably, much of this complexity stems from the paradox of object orientation, which seems to provide excellent paradigms for analysis and implementation, but present major difficulties for the designer."

- In 90s, attempts to develop other paradigms for transferring design knowledge, e.g., patterns and architectures.

- Peculiarity of software design: extent to which commercial interests have dominated the codifying of associated practices
  - More widely known design methods have been developed and marketed largely by consultants and commercial organizations.
  - Not true for requirements or testing
  - Suggests a real need for design skills, but does not create an objective forum for evaluation.

- Conclusions:
  - Life belt has become waterlogged and acting more like a leg iron.
  - Need to stop pretending that software design is largely a matter of following a set of well-defined activities. Recognize it as a creative process that requires us to develop design skills needed to build software systems of the future.
  - How do we identify, grow, and encourage those talents needed for the great designers who will create elegant and effective solutions to problems?