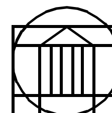
STRUCTURED ANALYSIS AND DESIGN

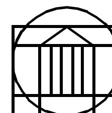


UVA

Department of Computer Science

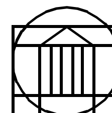
STRUCTURED METHODS

- *Structured methods* for analysis and design:
  - refer to a set of related methods from the late 1970s and 1980s;
  - were often useful for information systems or data processing system;
  - were extended for use with real-time system;
  - contributed a number of diagrams and notations that are important to know about;
  - varied greatly in the process or method on how to use these notations;
  - remain useful (to some extent) for non-object oriented development.
- Structured Analysis is a method for capturing system requirements and specifying them in a way that is useful for design.
- Structured Design is a design method for creating architectural, module and data design models.
- Both are based on:
  - function-based decomposition, and
  - top-down refinement.



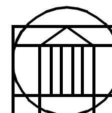
Methods and Religions

- What do we mean by a *method* or *methodology* in this context?
  - a *process*, or a series of steps to carry out to accomplish some task in the development life-cycle (e.g. requirements specification);
  - a set of *notations* or *products* which form a *model* of the system;
  - and, a set of rules on how to use the model.
- A somewhat silly analogy: methods are like religions sometimes!
  - Both are often prescriptive: follow these rules to achieve some goals. Bad things will happen if you don't!
  - Both usually have important symbols or representations that we use in following the rules.
  - There are related families of religions and methods with a lot in common.
  - Often both are "founded" by someone, whose name is attached to the religion or method. (Methods: Yourdon/DeMarco, Shlaer/Mellor, Booch, Rumbaugh, etc.)
  - Finally, people can be fanatical about both! (Advice: keep an open mind.)



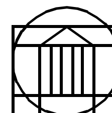
STRUCTURED DESIGN

- **Goal:** Describe system design at both architectural level and module level.
- **Approach:** top-down functional decomposition to model modules and their interfaces.
- **Notations/Work Products:**
  - Structured charts: Show modules as a call-tree, with invocations and interface.
  - Module descriptions: descriptions of how to implement individual modules.
    - Pseudo-code is a good choice.
  - Data dictionary, with complete implementation details for majors data items in the system (files, parameters, globals, data structures, etc.)
- **Process:** The method provides guidelines or heuristics for moving from Structured Analysis models created earlier in the life-cycle to design models.
  - Two useful approaches: *transform analysis* and *transaction analysis*. (Jalote, pp. 236-238, 228-221.)
  - Practitioners have struggled with moving from analysis to design with structured methods. OO methods often said to be more integrated in this regard.
- But, Structure Charts are a fine approach for modeling non-OO designs.

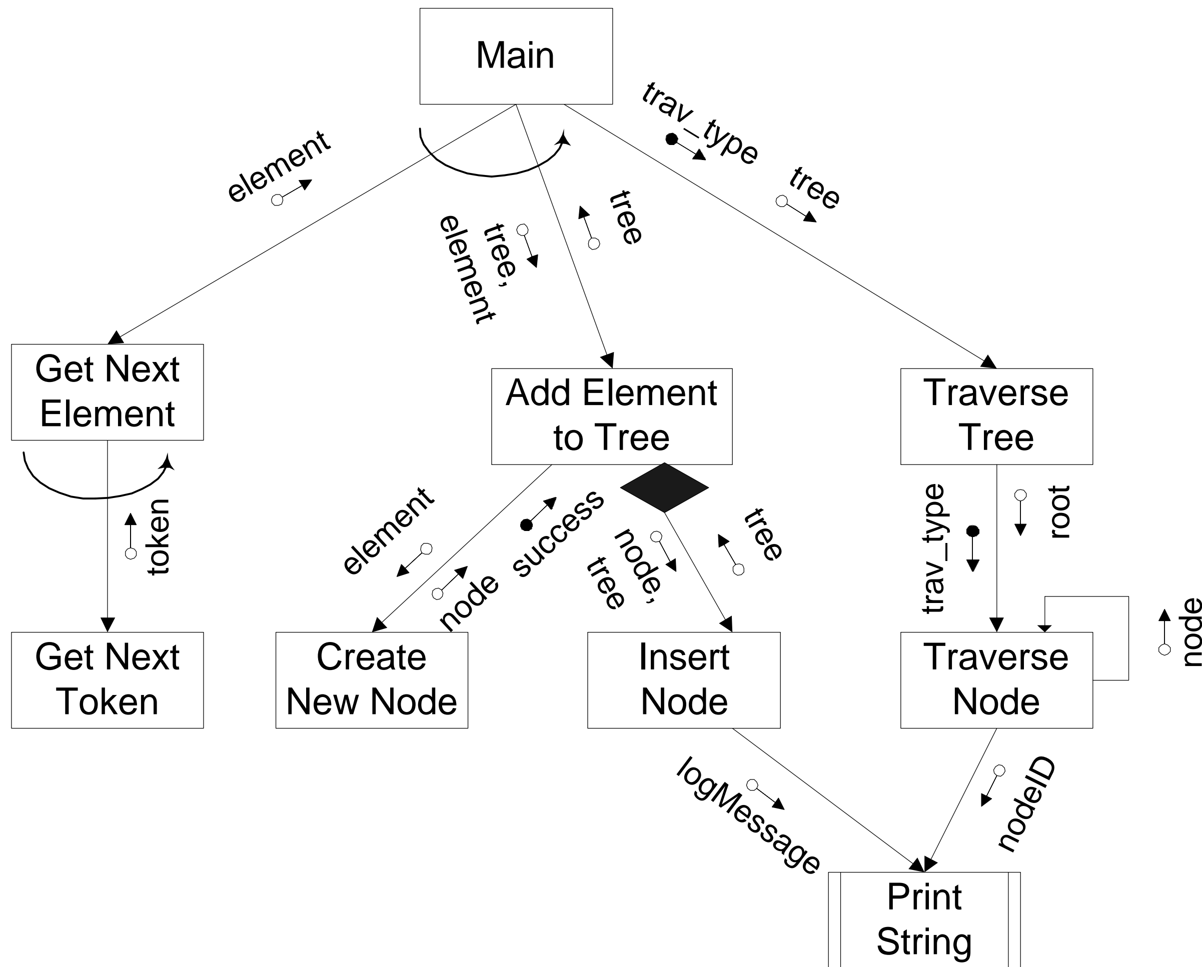


COMPONENTS OF A STRUCTURE CHART

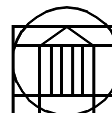
- Modules: Drawn as rectangles with module name inside.
  - Module name will probably be used in implementation.
  - Also create a *module specification* for each module, showing the logic that implements the module. This is often pseudo-code.
- Calling tree: Each module called by a module is shown as a child in the tree.
  - Module X called twice by Module Y is usually only shown *once* below Y. Children are shown in order they're called.
- Interfaces Between Modules: Show all inputs and outputs for each module pair.
  - *Data couples*: arrow with hollow circle
  - *Flag*: arrow with solid circle
  - Arguments that are both input and output may have bi-directional arrows.
- Library modules:
  - Rectangles with double-vertical lines on each side.
  - Code you won't write because it's reused, standard library, etc.
  - Library modules often omitted from diagrams when frequent.



STRUCTURE CHART EXAMPLE



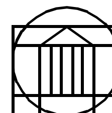
- Note: separate structure charts may show lower-level detail (perhaps a subtree rooted at GetNextToken) or reused subtrees.



UVA
Department of Computer Science

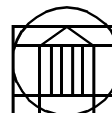
COMMENTS ON THE EXAMPLE STRUCTURE CHART

- The design is for a program that:
  - Builds a *tree*, where *elements* are stored in *nodes*, and elements are assembled from *tokens*.
  - The final tree is traversed in-order, pre-order, or post-order (perhaps according to a user-selected run-time parameter).
- Optional Structure Chart Notations:
  - *Repeated invocations* (i.e. loops): shown by an arc-arrow spanning some parent-child connections.
    - In example: multiple tokens are read to get one element. Also, multiple elements obtained and inserted before traversing the tree one time.
  - *Conditional invocation:* shown by diamond on parent-child connection. (E.g. if/then, case statements, etc.)
    - In example: a node isn't inserted into tree if it can't be created.
  - You can leave these notations off because each module has a mo*dule specification* which will show this in the pseudo-code.
  - Other optional notations include:
    - *Off-page connectors* that point to another structure chart.
    - *Information clusters*: a shared data object and modules that can access it.



Evaluating a Structure Chart

- Some design issues from inspecting this example structure chart:
  - There's control coupling between Main and TraverseTree. OK or preventable?
  - Should AddElementToTree return a flag indicating success or failure?
  - Should TraverseTree always be called? (The design indicates it is.)
  - Do we want to put GetNextElement and AddElementToTree into a module called BuildTree, and let Main call that? (That's a good idea!)
- Examples of general issues to look for when reviewing a Structure Chart:
  - Coupling and cohesion issues.
  - Common logic that could be put into a module and reused within the system.
  - Modules that do too much, or trivial modules, or redundant modules.
  - *Tramp data*: information that is passed through too many hands before getting where it's needed.
  - *Decision splits*: too much "distance" between when a condition is recognized and when it's handled. (Look for flags that travel too far.)
  - Overall architecture: Do the "high-level" modules handle data entities in their "highest" or most "abstract" form? Can information hiding be improved? How robust is the architecture to requirements change?



FINAL COMMENTS ON STRUCTURED METHODS

Final Comments on Structured Design:

- Again, there is a more detailed process for creating them, refining them:
  - Jalote, pp. 236-238, 228-221.)
  - Page-Jones, *The Practical Guide to Structured Systems Design* (1988). (The best reference for using structure charts well.)
- Structure charts are not the complete design:
  - *Data dictionary:* shows all data couples, flags, files, common data, etc. used in the structure chart, with all implementation details.
  - *Module specifications:* detailed design activity creates pseudo-code for each module after architecture is reviewed and signed off.

Final Comments on Structured Methods In General:

- Both DFDs and SCs model multiple levels of abstraction very well.
- These are good approaches if OO methods are not appropriate.
- But OO programming and design promote encapsulation and information hiding much better than a function-oriented approach.