SE 2A04 Fall 2001

Software Structure

Instructor: W. M. Farmer

Revised: 11 November 2001

Importance of Structure

- A good software product requires a good structure
- Several kinds of structure can be associated with a software product
  - Some structures are **hierarchical** (i.e., they can be represented by a directed acyclic graph (DAG))
  - Not all structures are equally important for a particular software product
  - Different structures may conflict with each other

Kinds of Software Structure

1. Data flow
2. Entity relationship
3. State transition
4. Abstraction
5. "Uses"
6. Access
7. File
8. Code

Data Flow Structure

- How does data flow through the product?
  How are outputs connected to inputs?
- Important when data flow is key
- **Data flow diagrams** are used to graphically represent the structure

Entity Relationship Structure

- What entities are part of the product?
  What relationships do the entities have?
- Important when data relationships are key
- **Entity-relationship diagrams** are used to graphically represent the structure

State Transition Structure

- What are the stable states of the product?
  What are the possible state transitions?
- Important when state is key
- **State transition diagrams** are used to graphically represent the structure
  - May not be practical if there are too many states

Abstraction Structure

- What serve as specifications in the product design?
  What serve as implementations in the product design?
  Where does refinement occur in the product design?
- The structure is usually hierarchical
- The structure includes the module structure
- **Abstraction diagrams** are used to graphically represent the structure
  - Shows the **satisfaction relation** between specifications and implementations

Uses Structure

- (Parnas) A procedure A with specification S **uses** a procedure B if A cannot satisfy S unless B is present and functioning correctly
  - A procedure A to calculate the average of a set of numbers uses a procedure B to do addition
  - A procedure B serving as a parameter of a procedure A may be called but is not used in the sense above
- Benefits of a well-designed **uses hierarchy**:
  - Product extension: procedures can be added without modifying the existing procedures
  - Product contraction: whole procedures can be deleted instead of modifying existing procedures
  - Characterization of possible subsets of the product
  - Hierarchy of languages

Criteria for Allowing a Procedure A to Use a Procedure B

1. A is simpler because it uses B
2. B is not more complex because it is not allowed to use A
3. There is a useful subset containing B and not A
4. There is no useful subset containing A and not B

References:

- D. Parnas, "Designing software for ease of extension and contraction", in: D. Hoffman and D. Weiss, *Software Fundamentals*, Addison Wesley, 2001.
- D. Parnas, "On a 'buzzword': hierarchical structure", in: D. Hoffman and D. Weiss, *Software Fundamentals*, Addison Wesley, 2001.

Access Structure

- **Subjects** are granted access privileges to **objects** on the basis of **trust**
  - Examples of subjects: Processes, procedures, OO objects, modules
  - Examples of objects: Variables, data structures, files, procedures, OO objects, modules
- Unauthorized access is either:
  - Made impossible or
  - Prevented by an **access control mechanism** which **authenticates** the subject and then checks whether it is **authorized** to access the object

File Structure: General Recommendations

- Express the structure of the software's design in the software's file structure
- Put files that work together in the same directory
- Use version control software to control and track modifications to files

Kinds of Files

- A software system will often contain various kinds of files for holding:
  - Source code
  - Object code
  - Scripts
  - Binary executables
  - Data
  - Documentation
- Use file name suffixes to distinguish between different kinds of files

Modules

- Put all the files associated with a module in the same directory
- The directory of a module should contain:
  - A **readme** file describing the module and its use
  - A **status** file listing what is finished and what needs to be done
  - An **install** file that will install the module
  - A **make** file to automatically update module files
  - A **maintenance** file explaining how to maintain the module files

Interfaces

- Put the interface and the implementation of a module in separate files or in separate parts of a file
  - Enables an implementation to be easily replaced
  - Other modules only need access to the interface file
  - In C, the interface can be put in a header file while the implementation is put in a source file
- List at the top of each implementation file the interfaces that the implementation uses
  - In C, this is done with an `#include` command

Code Structure: General Recommendations

- Be consistent
- As a general rule, choose clarity before efficiency
- Express the structure of the software's design in the software's code
- Follow the conventions of the programming language being used

Keep the Code Simple

- Write procedures that fit on one screen
- Put at most one programming statement on a line
- Keep the following measures low:
  - Loop nesting level
  - Conditional nesting level
  - Number of local variables in a procedure
- Avoid control structures that radically change state
  - Exits, gotos, state jumps, self-modifying code
- Avoid nonstandard language features

Naming Programming Entities

- Naming is an important but difficult task
- One should employ a naming convention
  - Names should be short and descriptive
  - The more global the entity, the more descriptive the name should be
  - The more local, the shorter the name can be
- A name may include:
  - Type of entity or return value
  - Name of module
- Words in a name can be separated by underscores, hyphens, and case changes, but avoid using spaces

Formatting Code

- Use formatting to display the structure of the code
  - Indentation to display subordinate relationships between code
  - Alignment to identify blocks of code
  - Blank lines to separate blocks of code
- Write fully bracketed code to facilitate maintenance
- Write code in tabular form whenever possible
- Avoid “wrap-around” code
- Line up comments to the right of the code

Scope of Variables

- Make the scope of variables as narrow as possible
  - Avoid global variables
- A wide-scoped variable is:
  - Harder to maintain because its instances may appear far apart from each other
  - More easily corrupted because its data can be modified by diverse procedures
- Decrease the scope of a variable by introducing procedures for accessing the variable

Procedures

- Use a convention for naming and ordering parameters
- Make explicit and carefully control any side-effects
  - Keep the use of side-effects to a minimum
- Make the scope of procedures as narrow as possible
- Any code fragment used more than once should be made into a procedure
  - Make procedures powerful
  - Use simple procedures to invoke powerful procedures in special ways

Code Documentation

- Components:
  - Specification of what the code is required to do
  - Pseudocode description of what the code does
  - Commented code
  - Proof that code's behavior satisfies its specification
  - Mapping of code specification back to the design
- Several approaches:
  - Generate documentation from code files
  - Generate code from documentation files
  - Generate documentation and code from common files

Commenting Code

- Begin every code file with:
  - Copyright statement
  - Authors
  - Description of contents
  - Revision date and log of changes made to the file
- Comment:
  - Each variable declaration
  - Each procedure definition
  - Loops and larger blocks of code
  - Anything that is not obvious
- Avoid excessive comments in procedure bodies
  - **Write code so that what it does is obvious**

Loops

- A loop terminates if there is a **natural number value** that **strictly decreases** with each iteration of the loop
- An **invariant** of a loop is a formula φ such that:
  - φ is true before the loop is executed
  - φ is true after each execution of the body of the loop
- The documentation of each loop should include:
  - A strictly decreasing natural number value
  - A loop invariant
- Ideally, the strictly decreasing natural number value and the invariant should be formulated before the loop is coded

Min and Max of an Array: Problem

- Let

$$\text{MinMax}\colon \text{Array}[1,n](\mathbf{Z}) \to \mathbf{N} \times \mathbf{N}$$

be the function that, given an array $a \in \text{Array}[1,n](\mathbf{Z})$, returns a pair (i,j) of indices of a such that

$$\forall m\colon \mathbf{N}\,.\; 1 \le m \le n \Rightarrow a[i] \le a[m] \le a[j]$$

- Problem: Implement MinMax

Min and Max of an Array: Solution

- procedure MinMax(a : Array$[1,n]$($\mathbf{Z}$)) : $\mathbf{N} \times \mathbf{N}$

```
  i, j, k : N;
  i, j ⇐ 1; k ⇐ 0;
  it
    k ⇐ k + 1;
    ((k ≤ n → (a[k] < a[i] → i ⇐ k |
               a[k] > a[j] → j ⇐ k |
               a[i] ≤ a[k] ≤ a[j] → skip)) |
     (k > n → skip));
    ((k < n → go) | (k ≥ n → stop))
  ti;
  return (i, j)
end procedure
```

- **Strictly decreasing natural number value**: $n - k$
- **Loop invariant**: $\forall m : \mathbf{N} \,.\, 1 \le m \le k \Rightarrow a[i] \le a[m] \le a[j]$

Euclid's GCD Algorithm: Problem

- The GCD of two positive integers is the **greatest common divisor** of the two integers
- Problem: Implement the function GCD : $\mathbf{Z} \times \mathbf{Z} \to \mathbf{Z}$
- Some mathematical facts:
  - If $x > 0$, $y > 0$, and $x > y$, then GCD$(x - y, y) =$ GCD(x, y)
  - If $x > 0$, then GCD$(x, x) = x$

Euclid's GCD Algorithm: Solution

- procedure GCD(x:$\mathbf{Z}$,y:$\mathbf{Z}$):$\mathbf{Z}$

```
  ((x > 0 ∧ y > 0 →
     it
       ((x > y → (x ⇐ x − y; go)) |
        (y > x → (y ⇐ y − x; go)) |
        (x = y → stop))
     ti) |
   (x ≤ 0 ∨ y ≤ 0 → error));
  return x
end procedure
```

- **Strictly decreasing natural number value**: max(x, y)
- **Loop invariant**: max$(x, y) \ge$ GCD$(x, y) =$ GCD(x_0, y_0)

Error Messages

- Make error messages as informative as possible
  - Indicate where in the code the error occurred
  - Describe the situation that caused the error
- "Throw" lower-level errors to appropriate higher-level code
- Write error messages for both the user and the developer

Coding Structure: Conclusions

- Use an effective coding style
- Continuously look for ways of making your code:
  - Simpler
  - More powerful
  - Better documented
- Make the structure of the software explicit