

Software Metrics and Design Principles

Chapters 5,8

What is Design?

- Design is the process of creating a plan or blueprint to follow during actual construction
- Design is a problem-solving activity that is iterative in nature
- In traditional software engineering the outcome of design is the **design document** or **technical specification** (if emphasis on notation)

“Wicked Problem”

- Software design is a “Wicked Problem”
 - Design phase can’t be solved in isolation
 - Designer will likely need to interact with users for requirements, programmers for implementation
 - No stopping rule
 - How do we know when the solution is reached?
 - Solutions are not true or false
 - Large number of tradeoffs to consider, many acceptable solutions
 - Wicked problems are a symptom of another problem
 - Resolving one problem may result in a new problem elsewhere; software is not continuous

Systems-Oriented Approach

- The central question: how to decompose a system into parts such that each part has lower complexity than the system as a whole, while the parts together solve the user’s problem?
- In addition, the interactions between the components should not be too complicated
- Vast number of design methods exist

Design Considerations

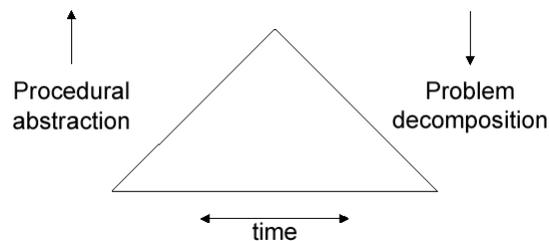
- “Module” used often – usually refers to a method or class
- In the decomposition we are interested in properties that make the system flexible, maintainable, reusable
 - Abstraction
 - Modularity
 - Information Hiding
 - Complexity
 - System Structure

Abstraction

- Abstraction
 - Concentrate on the essential features and ignore, abstract from, details that are not relevant at the level we are currently working on
 - E.g. Sorting Module
 - Consider inputs, outputs, ignore details of the algorithms until later
 - Two general types of abstraction
 - Procedural Abstraction
 - Data Abstraction

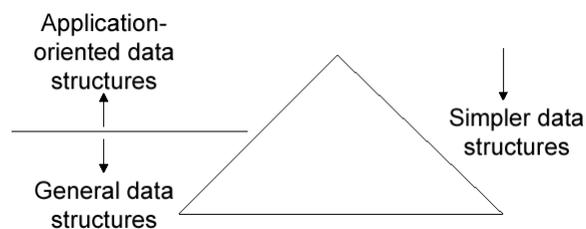
Procedural Abstraction

- Fairly traditional notion
 - Decompose problem into sub-problems, which are each handled in turn, perhaps decomposing further into a hierarchy
 - Methods may comprise the sub-problems and sub-modules, often in time



Data Abstraction

- From primitive to complex to abstract data types
 - E.g. Integers to Binary Tree to Data Store for Employee Records
- Find hierarchy in the data



Modularity

- During design the system is decomposed into modules and the relationships among modules are indicated
- Two structural design criteria as to the “goodness” of a module
 - Cohesion : Glue for intra-module components
 - Coupling : Strength of inter-module connections

Levels of Cohesion

1. Coincidental
 - Components grouped in a haphazard way
2. Logical
 - Tasks are logically related; e.g. all input routines. Routines do not invoke one another.
3. Temporal
 - Initialization routines; components independent but activated about the same time
4. Procedural
 - Components that execute in some order
5. Communicational
 - Components operate on the same external data
6. Sequential
 - Output of one component serves as input to the next component
7. Functional
 - All components contribute to one single function of the module
 - Often transforms data into some output format

Using Program and Data Slices to Measure Program Cohesion

- Bieman and Ott introduced a measure of program cohesion using the following concepts from program and data slices:
 - A data token is any variable or constant in the module
 - A slice within a module is the collection of all the statements that can affect the value of some specific variable of interest.
 - A data slice is the collection of all the data tokens in the slice that will affect the value of a specific variable of interest.
 - Glue tokens are the data tokens in the module that lie in more than one data slice.
 - Super glue tokens are the data tokens in the module that lie in every data slice of the program

Measure Program Cohesion through 2 metrics:

- weak functional cohesion = (# of glue tokens) / (total # of data tokens)
- strong functional cohesion = (#of super glue tokens) / (total # of data tokens)

Procedure Sum and Product

```
(N : Integer;  
  Var SumN, ProdN : Integer);  
Var      I : Integer  
Begin  
  SumN   := 0;  
  ProdN  := 1;  
  For I  := 1 to N do begin  
    SumN := SumN + I  
    ProdN := ProdN * I  
  End;  
End;
```

1-12

Data Slice for SumN

```
(N : Integer;  
  Var SumN, ProdN : Integer);  
  Var I : Integer  
  Begin  
    SumN := 0;  
    ProdN := 1;  
    For I := 1 to N do begin  
      SumN := SumN + I  
      ProdN := ProdN * I  
    End;  
  End;
```

Data Slice for SumN = $N_1 \cdot \text{SumN}_1 \cdot I_1 \cdot \text{SumN}_2 \cdot 0_1 \cdot I_2 \cdot 1_2 \cdot N_2 \cdot \text{SumN}_3 \cdot \text{SumN}_4 \cdot I_3$

1-13

Data Slice for ProdN

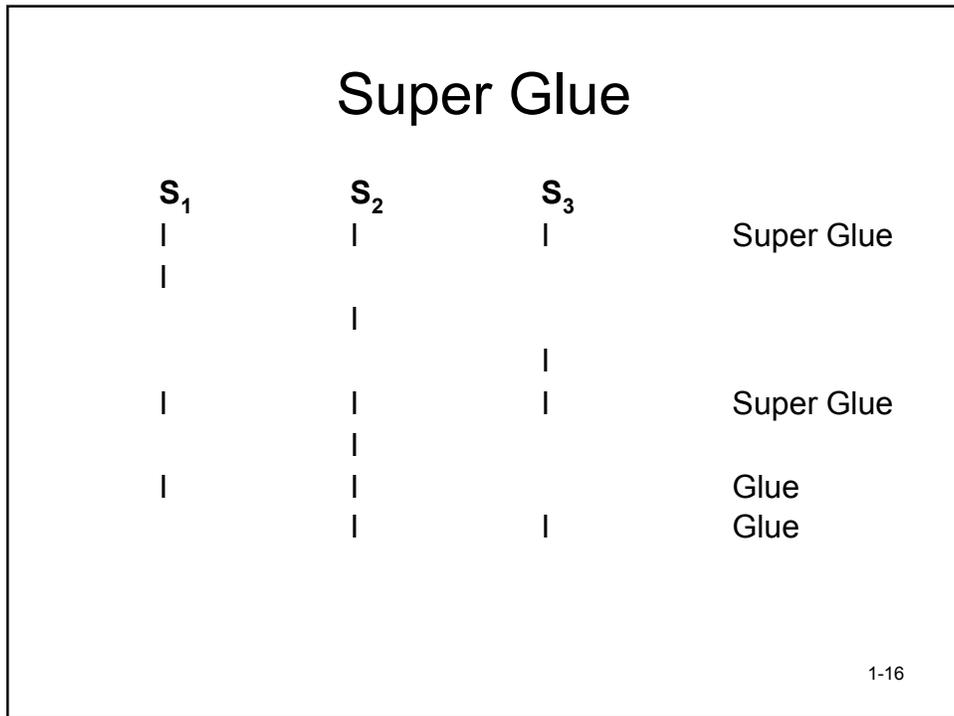
```
(N : Integer;  
  Var SumN, ProdN : Integer);  
  Var I : Integer  
  Begin  
    SumN := 0;  
    ProdN := 1;  
    For I := 1 to N do begin  
      SumN := SumN + I  
      ProdN := ProdN * I  
    End;  
  End;
```

Data Slice for ProdN = $N_1 \cdot \text{ProdN}_1 \cdot I_1 \cdot \text{ProdN}_2 \cdot 1_1 \cdot I_2 \cdot 1_2 \cdot N_2 \cdot \text{ProdN}_3 \cdot \text{ProdN}_4 \cdot I_4$

1-14

Data token	SumN	ProdN
N_1	1	1
Sum N_1	1	
Prod N_1		1
I_1	1	1
Sum N_2	1	
O_1	1	
Prod N_2		1
I_1		1
I_2	1	1
I_2	1	1
N_2	1	1
Sum N_3	1	
Sum N_4	1	
I_3	1	
Prod N_3		1
Prod N_4		1
I_4		1

1-15



Functional Cohesion

- Strong functional cohesion (SFC) in this case is the same as WFC
 $SFC = 5/17 = 0.294$
- If we had computed only SumN or ProdN then
 $SFC = 17/17 = 1$

1-17

Coupling

- Measure of the strength of inter-module connections
- High coupling indicates strong dependence between modules
 - Should study modules as a pair
 - Change to one module may ripple to the next
- Loose coupling indicates independent modules
 - Generally we desire loose coupling, easier to comprehend and adapt

Types of Coupling

1. Content
 - One module directly affects the workings of another
 - Occurs when a module changes another module's data
 - Generally should be avoided
2. Common
 - Two modules have shared data, e.g. global variables
3. External
 - Modules communicate through an external medium, like a file
4. Control
 - One module directs the execution of another by passing control information (e.g. via flags)
5. Stamp
 - Complete data structures or objects are passed from one module to another
6. Data
 - Only simple data is passed between modules

Modern Coupling

- Modern programming languages allow private, protected, public access
- Coupling may be modified to indicate levels of visibility, whether coupling is commutative
- Simple Interfaces generally desired
 - Weak coupling and strong cohesion
 - Communication between programmers simpler
 - Correctness easier to derive
 - Less likely that changes will propagate to other modules
 - Reusability increased
 - Comprehensibility increased

Dharma (1995)

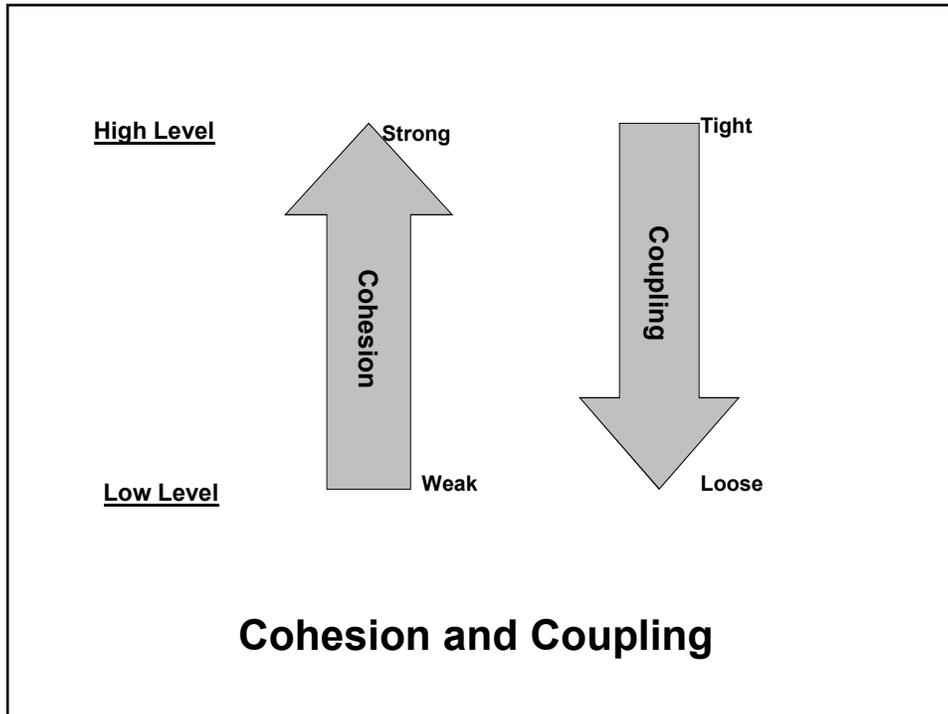
- Data and control flow coupling
 - d_i = number of input data parameters
 - c_i = number of input control parameters
 - d_o = number of output data parameters
 - c_o = number of output control parameters
- Global coupling
 - g_d = number of global variables used as data
 - g_c = number of global variables used as control
- Environmental coupling
 - w = number of modules called (fan-out)
 - r = number of modules calling the module under consideration (fan-in)

1-21

Dharma (1995)

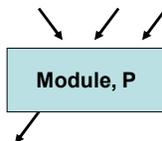
- Coupling metric (m_c)
 $m_c = k/M$, where $k = 1$
 $M = d_i + a * c_i + d_o + b * c_o + g_d + c * g_c + w + r$
where $a=b=c=2$
- The more situations encountered, the greater the coupling, and the smaller m_c
- One problem is parameters and calling counts don't guarantee the module is linked to the inner workings of other modules

1-22



Henry-Kafura (Fan-in and Fan-out)

- Henry and Kafura metric measures the inter-modular flow, which includes:
 - Parameter passing
 - Global variable access
 - inputs
 - outputs
- Fan-in : number of inter-modular flow into a program
- Fan-out: number of inter-modular flow out of a program



Module's Complexity, $C_p = (\text{fan-in} \times \text{fan-out})^2$
 for example above: $C_p = (3 + 1)^2 = 16$

Information Hiding

- Each module has a secret that it hides from other modules
 - Secret might be inner-workings of an algorithm
 - Secret might be data structures
- By hiding the secret, changes do not permeate the module's boundary, thereby
 - Decreasing the coupling between that module and its environment
 - Increasing abstraction
 - Increasing cohesion (the secret binds the parts of a module)
- Design involves a series of decisions. For each such decision, questions are: who needs to know about these decisions? And who can be kept in the dark?

Complexity

- Complexity refers to attributes of software that affect the effort needed to construct or change a piece of software
 - Internal attributes; need not execute the software to determine their values
- Many different metrics exist to measure complexity
- Two broad classes
 - Intra-Modular attributes
 - Inter-Modular attributes

Intra-Modular Complexity

- Two types of intra-modular attributes
 - Size-Based Metrics
 - E.g. Lines of Code
 - Obvious objections but still commonly used
 - Structure-Based Metrics
 - E.g. complexity of control or data structures

Halstead's Software Science

- Size-based metric
- Uses number of operators and operands in a piece of software
 - n_1 is the number of unique operators
 - n_2 is the number of unique operands
 - N_1 is the total number of occurrences of operators
 - N_2 is the total number of occurrences of operands
- Halstead derives various entities
 - Size of Vocabulary: $n = n_1 + n_2$
 - Program Length: $N = N_1 + N_2$
 - Program Volume: $V = N \log_2 n$
 - Visualized as the number of bits it would take to encode the program being measured

Halstead's Software Science

- Potential Volume: $V^* = (2+n_2)\log(2+n_2)$
 - V^* is the volume for the most compact representation for the algorithm, assuming only two operators: the name of the function and a grouping operator. n_2 is minimal number of operands.
- Program Level: $L = V^*/V$
- Programming Effort: $E = V/L$
- Programming Time in Seconds: $T = E/18$
- Numbers derived empirically, also based on speed human memory processes sensory input

Halstead metrics really only measures the lexical complexity, rather than structural complexity of source code.

Software Science Example

```
1. procedure sort(var x:array; n: integer)
2.     var i,j,save:integer;
3.     begin
4.         for i:=2 to n do
5.             for j:=1 to i do
6.                 if x[i]<x[j] then
7.                     begin save:=x[i];
8.                         x[i]:=x[j];
9.                         x[j]:=save
10.                    end
11.     end
```

Software Science Example

Operator	#
procedure	1
sort()	1
var	2
:	3
array	1
;	6
integer	2
,	2
begin...end	2
for..do	2
if...then	1
:=	5
<	1
□	6
n1=14	N1=35

Operand	#
x	7
n	2
i	6
j	5
save	3
2	1
1	1
n2=7	N2=25

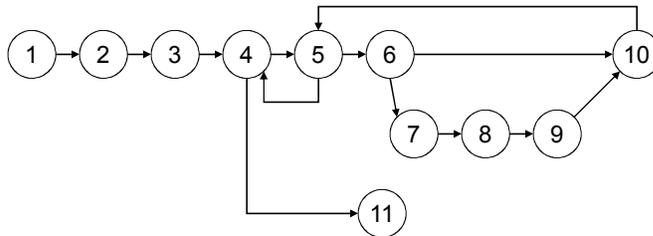
Size of vocabulary: 21
 Program length: 60
 Program volume: 264
 Program level: 0.04
 Programming effort: 6000
 Estimated time: 333 seconds

Structure-Based Complexity

- McCabe's Cyclomatic Complexity
- Create a directed graph depicting the control flow of the program
 - $CV = e - n + 2p$
 - CV = Cyclomatic Complexity
 - e = Edges
 - n = nodes
 - p = connected components

Cyclomatic Example

For Sorting Code; numbers refer to line numbers



$$CV = 13 - 11 + 2*1 = 4$$

McCabe suggests an upper limit of 10

- **T.J. McCabe's Cyclomatic complexity metric is based on the belief that program quality is related to the complexity of the program control flow.**

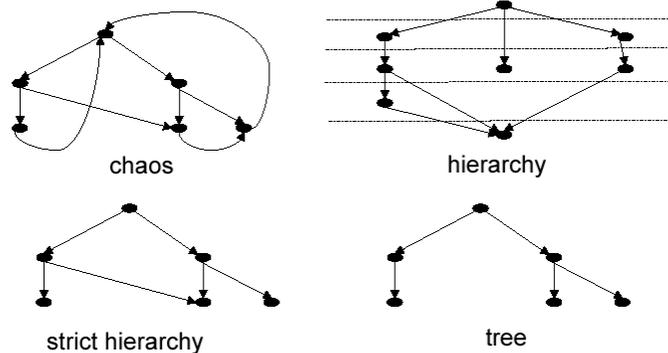
Shortcomings of Complexity Metrics

- Not context-sensitive
 - Any program with five if-statements has the same cyclomatic complexity
 - Measure only a few facts; e.g. Halstead's method doesn't consider control flow complexity
- Others?
- Minix:
 - Of the 277 modules, 34 have a CV > 10
 - Highest has 58; handles ASCII escape sequences. A review of the module was deemed "justifiably complex"; attempts to reduce complexity by splitting into modules would increase difficulty to understand and artificially reduce the CV

System Structure – Inter-Module Complexity

- The design may consist of modules and their relationships
- Can denote this in a graph; nodes are modules and edges are relationships between modules
- Types of inter-module relationships:
 - Module A contains Module B
 - Module A follows Module B
 - Module A delivers data to Module B
 - Module A uses Module B
- We are mostly interested in the last one, which manifests itself via a call graph
 - Possible shapes:
 - Chaotic
 - Directed Acyclic Graph (Hierarchy)
 - Layered Graph (Strict Hierarchy)
 - Tree

Module Hierarchies



Graph Metrics

- Metrics use:
 - Size of the graph
 - Depth
 - Width (maximum number of nodes at some level)
- A tree-like call graph is considered the best design
 - Some metrics measure the deviation from a tree; the **tree impurity** of the graph
 - Compute number of edges that must be removed from the graph's minimum spanning tree
- Other metrics
 - $\text{Complexity}(M) = \text{fanin}(M) * \text{fanout}(M)$
 - Fanin/Fanout = local and global data flows

Software Metrics Etiquette

- Use common sense and organizational sensitivity when interpreting metrics data.
- Provide regular feedback to the individuals and teams who have worked to collect measures and metrics.
- Don't use metrics to appraise individuals
- Work with practitioners and teams to set clear goals and metrics that will be used to achieve them.
- Never use metrics to threaten individuals or teams.
- Metrics data that indicate a problem area should not be considered "negative". These data are merely an indicator for process improvement.
- Don't obsess on a single metric to the exclusion of other important metrics.

1-38