SENG 421: Software Metrics

Measuring Internal Product Attributes: Structural Complexity (Chapter 6)

Department of Electrical & Computer Engineering, University of Calgary

B.H. Far (far@ucalgary.ca)

http://www.enel.ucalgary.ca/People/far/Lectures/SENG421/06/
Problem Statement

How complex is the following program?

```plaintext
1: read x,y,z;
2: type = "scalene";
3: if (x == y or x == z or y == z) type = "isosceles";
4: if (x == y and x == z) type = "equilateral";
5: if (x >= y+z or y >= x+z or z >= x+y) type = "not a triangle";
6: if (x <= 0 or y <= 0 or z <= 0) type = "bad inputs";
7: print type;
```

Is there a way to measure it?

It has something to do with program structure (branches, nesting) & flow of data
Contents

- Software structural measurement
- Control-flow structure
- Structural complexity: cyclomatic complexity
- Data flow and data structure attributes
- Architectural measurement
Software Complexity Metrics

- Depth of nesting
- Cyclomatic complexity
- Morphological measures
- Cohesion
- Coupling
- Information flow complexity
- Data structure complexity

Code
Architecture
How to Represent Program Structure?

Software structure can have 3 attributes:

- **Control-flow structure:** Sequence of execution of instructions of the program.
- **Data flow:** Keeping track of data as it is created or handled by the program.
- **Data structure:** The organization of data itself independent of the program.
Q1: How to represent “structure” of a program?

A1: Control-flow diagram

Q2: How to define “complexity” in terms of the structure?

A2: Cyclomatic complexity; depth of nesting
Basic Control Structures (BCSs) are set of essential control-flow mechanisms used for building the logical structure of the program.

BCS types:

- **Sequence:** e.g., a list of instructions with no other BCSs involved.
- **Selection:** e.g., if ... then ... else.
- **Iteration:** e.g., do ... while; for ... to ... do.
There are other types of BCSs, (may be called advanced BCSs), such as:

- Procedure/function/agent call
- Recursion (self-call)
- Interrupt
- Concurrence
Control Flow Graph (CFG)

- Control flow structure is usually modeled by a directed graph (di-graph)
  \[
  \text{CFG} = \{N, A\}
  \]
- Each node \( n \) in the set of nodes (\( N \)) corresponds to a program statement.
- Each directed arc (or directed edge) \( a \) in the set of arcs (\( A \)) indicates flow of control from one statement of program to another.
  - **Procedure nodes**: nodes with out-degree 1.
  - **Predicate nodes**: nodes with out-degree other than 1 and 0.
  - **Start node**: nodes with in-degree 0.
  - **Terminal (end) nodes**: nodes with out-degree 0.
Control Flow Graph (CFG) /2

Definition [FeR97]:
- A flowgraph is a directed graph in which two nodes, the start node and the stop node, obey special properties: the stop node has out-degree zero, and the start node has in-degree zero. Every node lies on some path from the start node to the stop node.
Example 1

\[
\begin{align*}
S &= 0 \\
i &= 1 \\
\text{if } i < (n+1) \text{ then} & \\
\text{yes: } S &= S + a_i \\
i &= i + 1 \\
\text{no: } & \\
\text{end } \\
\end{align*}
\]

Print S
if a then
  if b then X Y
  while e do U
else
  if c then
    repeat V until d
  endif
endif
The control flow graph $\text{CFG} = \{N, A\}$ model for a program does not explicitly indicate how the control is transferred. The finite-state machine (FSM) model for $\text{CFG}$ does.

$$M = \{N, \Sigma, \delta, n_0, F\}$$

$N$ set of nodes; $\Sigma$ set of input symbols (arcs)

$\delta$ transition function

$$\delta(p, a) = q ; \quad p, q \in N \quad \text{and} \quad a \in \Sigma$$

$n_0 \in N$ starting node and $F \subseteq N$ set of terminal node(s)
Example: FSM Model

- Finite state machine model for the increment and add example
Control Flow Graph /3

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Selection</th>
<th>Iteration</th>
<th>Procedure/function call</th>
<th>Recursion</th>
<th>Interrupt</th>
<th>Concurrence</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Sequence Diagram" /></td>
<td><img src="image" alt="Selection Diagram" /></td>
<td><img src="image" alt="Iteration Diagram" /></td>
<td><img src="image" alt="Procedure/function call Diagram" /></td>
<td><img src="image" alt="Recursion Diagram" /></td>
<td><img src="image" alt="Interrupt Diagram" /></td>
<td><img src="image" alt="Concurrence Diagram" /></td>
</tr>
</tbody>
</table>

---

far@ucalgary.ca
Common CFG Program Models

$P_n$ or $P_n(X_1,X_2,...,X_n)$

$X_1;X_2;...;X_n$

$D_0$ or $D_0(A, X)$

If $A$ then $X$

$D_1$ or $D_1(A, X, Y)$

If $A$ then $X$
else $Y$

$D_2$ or $D_2(A, X)$

while $A$ do $X$

$D_3$ or $D_3(A, X)$

repeat $X$ until $A$

$C_n$ or $C_n(A, X_1,...,X_n)$

case $A$ of
$a_1 : X_1$
a_2 : $X_2$

....
an : $X_n$
Prime Flow Graphs

Prime flow graphs are flow graphs that cannot be decomposed non-trivially by sequencing and nesting.

Examples (according to [FeP97]):
- \( P_n \) (sequence of \( n \) statements)
- \( D_0 \) (if-condition)
- \( D_1 \) (if-then-else-branching)
- \( D_2 \) (while-loop)
- \( D_3 \) (repeat-loop)
- \( C_n \) (case)
Let $F_1$ and $F_2$ be two flowgraphs. Then, the sequence of $F_1$ and $F_2$, (shown by $F_1; F_2$) is a flowgraph formed by merging the terminal node of $F_1$ with the start node of $F_2$. 

![Diagram showing sequencing and nesting of flowgraphs]
Let $F_1$ and $F_2$ be two flowgraphs. Then, the nesting of $F_2$ onto $F_1$ at $x$, shown by $F_1(F_2)$ is a flowgraph formed from $F_1$ by replacing the arc from $x$ with the whole of $F_2$. 
S-structured Graph

A family $S$ of prime flowgraphs is called **S-structured graph** (or **S-graph**) if it satisfies the following recursive rules:

1) Each member of $S$ is S-structured.

2) If $F$ and $G$ are S-structured flowgraphs, so is the sequences $F;G$ and nesting of $F(G)$.

3) No flowgraph is S-structured unless it can be generated by finite number of application of the above (step 2) rules.
S-Structured Graph – Example

- $S^D = \{P_1, D_0, D_2\}$
- The class of $S^D$-graphs is the class of flow graphs that is called D-structured (or simply: structured) in the Structured Programming literature
- Böhm and Jacopini (1966) have shown that every algorithm can be encoded as an $S^D$-graph (i.e., as a sequence or nesting of statements, if-conditions and while-loops)
- Although $S^D$ is sufficient in this respect, normally if-then-else structures ($D_1$) and repeat-loops ($D_3$) are included in $S^D$. 
Prime Decomposition

- Any flow graph can be *uniquely* decomposed into a hierarchy of sequencing and nesting primes, called “decomposition tree”. (Fenton and Whitty, 1991)
Hierarchical Measures

- The decomposition tree is enough to measure a number of program characteristics, including:
  - Nesting factor (depth of nesting)
  - Structural complexity
  - etc.

```plaintext
if a then
  begin
    if b then do X;
    Y;
    while e do U
  end
else
  if c then do
    repeat V until d
```
Depth of Nesting

Depth of nesting $n(F)$ for a flowgraph $F$ can be expressed in terms of:

**Primes:**

$n(P_1) = 0$; $n(P_2) = n(P_3) = ... = n(P_k) = 1$

$n(D_0) = n(D_1) = n(D_2) = n(D_3) = 1$

**Sequences:**

$n(F_1;F_2;...;F_k) = \max(n(F_1), n(F_2), ..., n(F_k))$

**Nesting:**

$n(F(F_1,F_2,...,F_k)) = 1 + \max(n(F_1), n(F_2), ..., n(F_k))$
Depth of Nesting /2

Example:

\[ F = D_1((D_0; P_1; D_2), D_0(D_3)) \]

\[ n(F) = 1 + \max(x_1, x_2) \]

\[ x_1 = \max(1, 0, 1) \]

\[ x_2 = 1 + \max(1) = 2 \]

\[ n(F) = 1 + \max(1, 2) \]

\[ n(F) = 3 \]

Smaller depth of nesting indicates less complexity in coding and testing.
Cyclomatic Complexity
Cyclomatic Complexity

- A program’s complexity can be measured by the cyclomatic number of the program flowgraph.
- The cyclomatic number can be calculated in 2 different ways:
  - Flowgraph-based
  - Code-based
For a program with the program flowgraph $G$, the cyclomatic complexity $v(G)$ is measured as:

$$v(G) = e - n + 2p$$

- $e$: number of edges
  - Representing branches and cycles
- $n$: number of nodes
  - Representing block of sequential code
- $p$: number of connected components
  - For a single component, $p=1$
For a program with the program flowgraph $G$, the cyclomatic complexity $v(G)$ is measured as:

$$v(G) = 1 + d$$

- $d$: number of predicate nodes (i.e., nodes with out-degree other than 1)
  - $d$ represents number of loops in the graph
  - or number of decision points in the program

i.e., The complexity of primes depends only on the predicates (decision points or BCSs) in them.
Cyclomatic Complexity: Example

\[ v(G) = e - n + 2p \]
\[ v(G) = 7 - 6 + 2 \times 1 \]
\[ v(G) = 3 \]

Or

\[ v(G) = 1 + d \]
\[ v(G) = 1 + 2 = 3 \]
#include <stdio.h>
main()
{
    int a;
    scanf("%d", &a);
    if ( a >= 10 )
    {
        if ( a < 20 ) printf("10 < a < 20 %d
", a);
        else printf("a >= 20 %d
", a);
    }
    else printf("a <= 10 %d
", a);
}

\[ v(G) = 1 + 2 = 3 \]
Example: Graph Based

\[ v(G) = 16 - 13 + 2 = 5 \]

or

\[ v(G) = 4 + 1 = 5 \]
Example 1

Determine cyclomatic complexity for the following Java program:

\[ v = 1 + d \]
\[ v = 1 + 6 = 7 \]
Example 2

Determine cyclomatic complexity for the following flow diagram:

\[ v = 1 + d \quad v = 1 + 2 = 3 \]

or

\[ v = e - n + 2 \]

\[ v = 11 - 10 + 2 = 3 \]
Example 3A

- Two functionally equivalent programs that are coded differently
- Calculate cyclomatic complexity for both

\[ V_A = 7 \]
\[ V_B = 1 \]

There is always a trade-off between control-flow and data structure. Programs with higher cyclomatic complexity usually have less complex data structure. Apparently program B requires more effort that program A.
Cyclomatic Complexity: Critics

- **Advantages:**
  - Objective measurement of complexity

- **Disadvantages:**
  - Can only be used at the component level
  - Two programs having the same cyclomatic complexity number may need different programming effort
  - Same requirements can be programmed in various ways with different cyclomatic complexities
  - Requires complete design or code visibility
A software system can be represented by a graph, \( S = \{N, R\} \).

Each node \( n \) in the set of nodes (\( N \)) corresponds to a subsystem.

Each edge \( r \) in the set of relations (\( R \)) indicates a relation (e.g., function call, etc.) between two subsystems.
Morphology

- Morphology refers to the overall shape of the software system architecture.
- It is characterized by:
  - **Size**: number of nodes and edges
  - **Depth**: longest path from the root to a leaf node
  - **Width**: number of nodes at any level
  - **Edge-to-node ratio**: connectivity density measure
Morphology: Example

Size:
- 12 nodes
- 15 edges

Depth: 3

Width: 6

\[ e/n = 1.25 \]
Tree Impurity /1

- The tree impurity measures $m(G)$ how much the graph is different from a tree
- The smaller $m(G)$ denotes the better design
Tree Impurity /2

- Tree impurity can be defined as:

\[ m(G) = \frac{\text{number of edges more than spanning tree}}{\text{maximum number of edges more than spanning tree}} \]

\[ m(G) = \frac{2(e - n + 1)}{(n-1)(n-2)} \]

- Example:

\[ m(G_1) = 0 \quad m(G_2) = 0.1 \quad m(G_3) = 0.2 \]

\[ m(G_4) = 1 \quad m(G_5) = 1 \quad m(G_6) = 1 \]
Complexity Measures: Cohesion
Components & Modules

- A **module** or **component** is a bounded sequence of program statements with an aggregate identifier.

- A module usually has two properties:
  - *Nearly-decomposability property*: the ratio of data communication within the module is much (at least 10 times) more than the communication with the outside.
  - *Compilability property*: a module should be separately compilable (at least theoretically).
Software Architecture

- A **modular** or **component-based** system is a system that all the elements are partitioned into different components.

- Components do not share any element. There are only relationships across the components.
A component-based system (CBS) can be defined by a graph, \( S = \{C, Re\} \).

Each node \( c \) in the set of nodes (C) corresponds to a component.

Each edge \( r \) in the set of relations (Re) indicates an external relationship between two components.
CBS: Code Complexity

- Code complexity of a component-based system is the sum of cyclomatic complexity of its components.

- Example:
  - $v(C_1) = v(C_3) = 2$
  - $v(C_2) = 1$
  - $v(G) = 2 + 2 + 1 = 5$
Cohesion describes how strongly related the responsibilities between design elements can be described.

The goal is to achieve “high cohesion”.

High cohesion between classes is when class responsibilities are highly related.
Cohesion

- **Cohesion** of a module is the extent to which its individual components are needed to perform some task.

- **Types of cohesion (7 types):**
  - **Functional:** The module performs a single function
  - **Sequential:** The module performs a sequence of functions
  - **Communicative:** The module performs multiple function on the same body of data
Cohesion /2

Types of cohesion (cont’d):

- **Procedural:** The module performs more than one function related to a certain software procedure.
- **Temporal:** The module performs more than one function and they must occur within the same time span.
- **Logical:** The module performs more than one function and they are related logically.
- **Coincidental:** The module performs more than one function and they are unrelated.
Examples: Cohesion

High package cohesion (functional cohesion)

Low class cohesion (coincidental and logical cohesion)
CBS: Cohesion

- **Cohesion** of a module is the extent to which its individual components are needed to perform some task.

- Cohesion for a component is defined in terms of the ratio of internal relationships to the total number of relationships.

\[
CH (C_i) = \frac{R_{\text{internal}}}{R_{\text{internal}} + R_{\text{external}}}
\]
CBS: System Cohesion

- **System cohesion** is the mathematical mean of cohesion of all its components.

\[
CH = \frac{1}{n} \sum_{i=1}^{n} CH(C_i) \quad 0 \leq CH \leq \%100
\]

- The higher system cohesion is better because it indicates that more job is processed internally.
Example: Cohesion

- Cohesion for a component is defined in terms of the ratio of internal relationships to the total number of relationships.

\[
CH(C_i) = \frac{R_{\text{internal}}}{R_{\text{internal}} + R_{\text{external}}}
\]

- Example:
  - \(CH(C1) = \frac{2}{3}\)
  - \(CH(C2) = \frac{1}{3}\)
  - \(CH(C3) = \frac{2}{3}\)
Example: System Cohesion

System cohesion is the mathematical mean of cohesion of all its components.

\[ CH = \frac{1}{n} \sum_{i=1}^{n} CH(C_i) \quad 0 \leq CH \leq \%100 \]

Example:

\[ CH = \frac{(2/3) + (1/3) + (2/3))}{3} = 5/9 = \%55 \]
Complexity Measures: Coupling
Concept: Coupling

- Coupling describes how strongly one element relates to another element.
- The goal is to achieve “loose coupling”.
- Loose coupling between classes is small, direct, visible, and has flexible relations with other classes.
Coupling: Package Dependencies

- Packages should not be cross-coupled.
- Packages in lower layers should not be dependent upon packages in upper layers.
- In general, dependencies should not skip layers (unless specified by the architecture).

\[ \equiv \text{Coupling violation} \]
Coupling: Class Relationships

- Strive for the loosest coupling possible

Strong Coupling

Loose Coupling
Coupling

Coupling is the **degree of interdependence** among modules.

Various types of coupling (5 types):

- **R0: independence**: modules have no communication
- **R1: data coupling**: modules communicate by parameters
- **R2: stamp coupling**: modules accept the same record type
- **R3: control coupling**: \( x \) passes the parameters to \( y \) and the parameter passed is a flag to control the behaviour of \( y \).
- **R4: content coupling**: \( x \) refers to inside of \( y \); branches into or changes data in \( y \).
Coupling /2

- No coupling: R0
- Loose coupling: R1 and R2
- Tight coupling: R3 and R4

There is no standard measurement for coupling!
**CBS: Coupling**

- **Coupling** of a component is the ratio of the number of external relations to the total number of relations.

\[
CP(C_i) = \frac{R_{\text{external}}}{R_{\text{internal}} + R_{\text{external}}}
\]
CBS: System Coupling

- **System coupling** is the mathematical mean of coupling of all its components.

\[ CP = \frac{1}{n} \sum_{i=1}^{n} CP(C_i) \quad 0 \leq CP \leq \%100 \]

- The lower system coupling is better because it indicates that less effort is needed externally.
Coupling in CBS: Example

- **Coupling** of a component

\[
CP(C_i) = \frac{R_{\text{external}}}{R_{\text{internal}} + R_{\text{external}}}
\]

- **Example:**
  - \( CP(C1) = 1/3 \)
  - \( CP(C2) = 2/3 \)
  - \( CP(C3) = 1/3 \)
CBS: System Coupling

- **System coupling** is the mathematical mean of coupling of all its components.

\[
CP = \frac{1}{n} \sum_{i=1}^{n} CP(C_i) \quad 0 \leq CP \leq %100
\]

- **Example:**
  - \( CP = \frac{(1/3)+(2/3)+(1/3)}{3} = 4/9 = %45 \)
Coupling: Representation

- Graph representation of coupling

- Coupling between modules $x$ and $y$:

$$c(x, y) = i + \frac{n}{n + 1}$$

- $i$: degree of worst coupling relation

- $n$: number of interconnections

Coupling type $R1 \sim R4$
Coupling: Representation /2

- Global coupling of a system consists of modules is the median value of the set of all couplings \( c(x,y) \)

\[
C(S) = Med \left\{ c(x, y) \mid \forall x, y \in S \right\}
\]

- Statistical review:
  - Median value: The median value is a measure of central tendency. The median value is the middle value in a set of values. Half of all values are smaller than the median value and half are larger. When the data set contains an odd (uneven) set of numbers, the middle value is the median value. When the data set contains an even set of numbers, the middle two numbers are added and the sum is divided by two. That number is the median value.
Example

- The structural modules of a software system and their interconnections are depicted below. The arrows depict the coupling between modules.
**Example (cont’d)**

a) Determine the coupling between modules.

b) Determine the global system coupling.

\[
\begin{align*}
C_{CH} & = 1 + 1/2 = 1.50 \\
C_{AE} = C_{BF} & = 1 + 2/3 = 1.66 \\
C_{AD} & = 2 + 1/2 = 2.50 \\
C_{AB} = C_{CD} = C_{CG} = C_{DJ} & = 2 + 2/3 = 2.66 \\
C_{BH} = C_{DI} = C_{EL} & = 3 + 1/2 = 3.50 \\
C_{EK} & = 3 + 3/4 = 3.75 \\
C_{AC} & = 4 + 1/2 = 4.50 \\
C_{BG} & = 4 + 2/3 = 4.66 \\
\end{align*}
\]

\[
C(S) = (2.66 + 2.66)/2 = 2.66
\]
Other Structural Metrics
Information flow is measured in terms of fan-in and fan-out of a component.

**Fan-in** of a module $M$ is the number of flows terminating at $M$ plus the number of data structures from which info is retrieved by $M$.

**Fan-out** of a module $M$ is the number of flows starting at $M$ plus the number of data structures updated by $M$. 
Information Flow Measures /2

- Information flow complexity (IFC) [Henry-Kafura, 1981]:

\[ IFC(M) = \text{length}(M) \times \left( \text{fan-in}(M) \times \text{fan-out}(M) \right)^2 \]

- IFC [Shepperd, 1990]:

\[ IFC(M) = \left( \text{fan-in}(M) \times \text{fan-out}(M) \right)^2 \]
Information Flow Measures /3

Example

<table>
<thead>
<tr>
<th>Module</th>
<th>fan-in</th>
<th>fan-out</th>
<th>([(\text{fan-in})(\text{fan-out})]^2)</th>
<th>length</th>
<th>‘complexity’</th>
</tr>
</thead>
<tbody>
<tr>
<td>WC</td>
<td>2</td>
<td>2</td>
<td>16</td>
<td>30</td>
<td>480</td>
</tr>
<tr>
<td>FD</td>
<td>2</td>
<td>2</td>
<td>16</td>
<td>11</td>
<td>176</td>
</tr>
<tr>
<td>CW</td>
<td>3</td>
<td>3</td>
<td>27</td>
<td>40</td>
<td>1080</td>
</tr>
<tr>
<td>DR</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>23</td>
<td>0</td>
</tr>
<tr>
<td>GDN</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td>RD</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>28</td>
<td>112</td>
</tr>
<tr>
<td>FWS</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>46</td>
<td>46</td>
</tr>
<tr>
<td>PW</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>29</td>
<td>29</td>
</tr>
</tbody>
</table>
Revision of the Henry-Kafura IFC measure:

- Recursive module calls should be treated as normal calls.
- Any variable shared by two or more modules should be treated as a global data structure.
- Compiler and library modules should be ignored.
- Indirect flow’s should be ignored.
- Duplicate flows should be ignored.
- Module length should be disregarded, as it is a separate attribute.
Data Structure Measurement

- There is always a trade-off between control-flow and data structure.
- Programs with higher cyclomatic complexity usually have less complex data structure.
- A simple example for data-structure measurement:

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Measurement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integers</td>
<td>$C_1 = n_i \times 1$</td>
</tr>
<tr>
<td>Strings</td>
<td>$C_2 = n_s \times 2$</td>
</tr>
<tr>
<td>Arrays</td>
<td>$C_3 = n_a \times 2 \times \text{size of array}$</td>
</tr>
<tr>
<td>Total</td>
<td>$C = C_1 + C_2 + C_3$</td>
</tr>
</tbody>
</table>
Example 3B

Calculate date structure complexity for Program A and B

<table>
<thead>
<tr>
<th></th>
<th>Program A</th>
<th>Program B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integers ($n_i$)</td>
<td>$C1 = 1 \times 1$</td>
<td>$C1 = 1 \times 1$</td>
</tr>
<tr>
<td>Strings ($n_s$)</td>
<td>$C2 = 6 \times 2$</td>
<td>$C2 = 6 \times 2$</td>
</tr>
<tr>
<td>Arrays ($n_a$)</td>
<td>$C3 = 0 \times 2 \times 0$</td>
<td>$C3 = 1 \times 2 \times 11$</td>
</tr>
<tr>
<td>Total</td>
<td>13</td>
<td>35</td>
</tr>
</tbody>
</table>

There is always a trade-off between control-flow and data structure. Programs with higher cyclomatic complexity usually have less complex data structure. Apparently program B requires more effort that program A.
Conclusion: Complexity Metrics

Complexity Metrics

- Depth of nesting
- Cyclomatic complexity
- Morphological measures
- Cohesion
- Coupling
- Information flow complexity
- Data structure complexity

Code
Architecture

far@ucalgary.ca