Software Metrics (ISAD3002/ISAD4002)

Lecture 3: Static Code Metrics

Updated: 26th May, 2017

Amy Affleck
Department of Computing
Curtin University

Outline

Static Metrics

- Metrics based on the structure of code that affect/measure.
  - Understandability;
  - Modularity;
  - Reuseability.
- Do not change during execution.

Cyclomatic Complexity

McCabe (1976)

- Developed in 1976, though still valid.
- Measures the complexity of a submodule or function.
- Can measure the complexity of a class by adding each functions complexity — Weighted Methods per Class (WMC).
- Based on the theory that decision making is inherently harder to understand.
  - Long code, sequential statements/calls to other methods. v.s.
  - Short code, numerous control structures/loops.
- Represents how difficult the code is to understand.
- Shares a basis with Software Testing graph coverage (white box).

Calculating Cyclomatic Complexity

- Frequently calculated/represented using a *Control Flow Graph*
  - Alternatively, count decision points in the code (can be error prone).
- There are two equations based on a CFG
- Method 1: count the number of decision points (d).

$$cc = d + 1$$

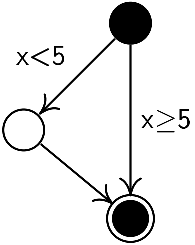
- Method 2: count the number of edges (e) and the number of nodes (n).

$$cc = e - n + 2$$

Cyclomatic Complexity

Selection - If & Ternary Operator

```
if( x < 5)
{
    ...
}
```



$$cc = e - n + 2 \qquad cc = d + 1$$

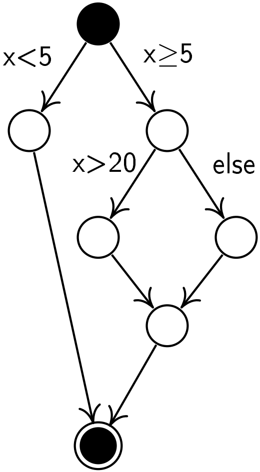
$$cc = 3 - 3 + 2 \qquad cc = 1 + 1$$

$$cc = 2 \qquad cc = 2$$

Cyclomatic Complexity

Selection — Nested If

```
if( x < 5){
    .. }
else{
    if (x > 20){
        ...  }
    else{
        ...}
}
```



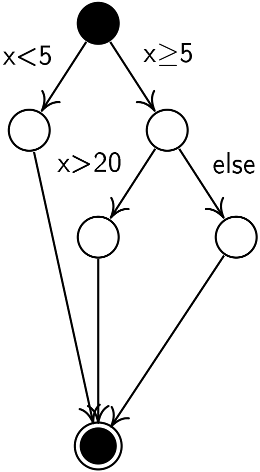
$$cc = e - n + 2$$
$$cc = 8 - 7 + 2$$
$$cc = 3$$

$$cc = d + 1$$
$$cc = 2 + 1$$
$$cc = 3$$

Cyclomatic Complexity

Selection - If-then-ElseIf

```
if( x < 5){
    ... }
else if (x > 20){
    ... }
else{
    ... }
```



$$cc = e - n + 2 \qquad cc = d + 1$$
$$cc = 7 - 6 + 2 \qquad cc = 2 + 1$$
$$cc = 3 \qquad cc = 3$$

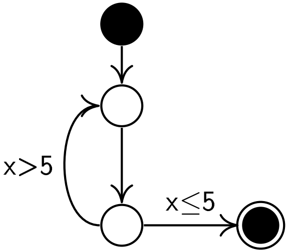
Cyclomatic Complexity

Loops — Do While

```
do
{
    ...
}while(x > 5);
```

$$cc = e - n + 2$$
$$cc = 4 - 4 + 2$$
$$cc = 2$$

$$cc = d + 1$$
$$cc = 1 + 1$$
$$cc = 2$$

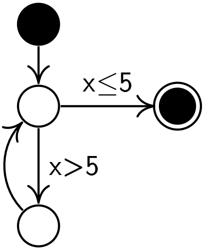


Cyclomatic Complexity

Loops — While/For

```
while(x > 5)
{
    ...
}
```

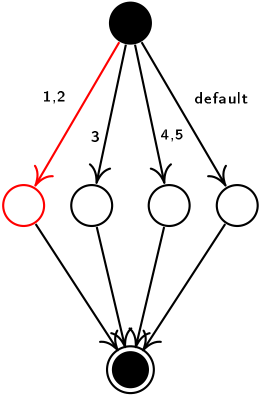
$$cc = e - n + 2 \qquad cc = d + 1$$
$$cc = 4 - 4 + 2 \qquad cc = 1 + 1$$
$$cc = 2 \qquad cc = 2$$



Cyclomatic Complexity

Selection — Case Statment

```
switch(x){
    case 1: case 2:
        ...
        break;
    case 3:
        ...
        break;
    case 4: case 5:
        ...
        break;
    default:
        ...  }
```



$$cc = e - n + 2$$
$$cc = 8 - 6 + 2$$
$$cc = 4$$

$$\cancel{cc = d + 1}$$
$$\cancel{cc = 1 + 1}$$
$$\cancel{cc = 2}$$

Cyclomatic Complexity

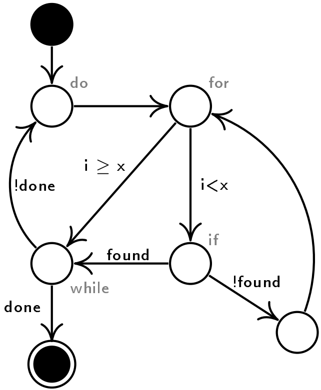
Nesting & Breaks

```
do{
    for(int i =0; i<x; i++){
        if(found(i)){
            num=i;
            break;
        }
      ...
    }
}while(!done);
```

$$cc = e - n + 2 \qquad cc = d + 1$$

$$cc = 9 - 7 + 2 \qquad cc = 3 + 1$$

$$cc = 4 \qquad cc = 4$$



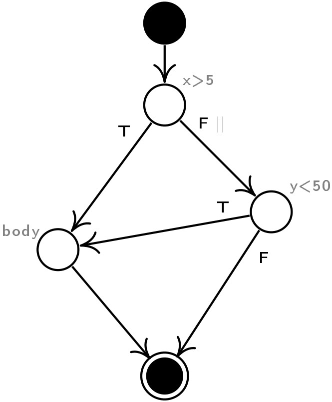
Cyclomatic Complexity

Short Circuit Evaluation

```
if (x > 5 || y < 50)
{
    ...
}
```

$$cc = e - n + 2$$
$$cc = 6 - 5 + 2$$
$$cc = 3$$

$$cc = d + 1$$
$$cc = 2 + 1$$
$$cc = 3$$



Cyclomatic Complexity

Notes

- CC does not just count *each* pathway, they must be linearly independent.
- Do not count try-catch statements.
  - Exceptions can be thrown at any node.
  - If you try the graphs will be unnecessarily complex.
  - CC reflects the difficulty of the code to be understood. Exceptions simplify code because we understand them.
  - i.e.: without try-catch you would have a significant number of control statements, this would increase complexity.
- Rule of Thumb, if CC is greater than 10-15 the module should be split.
- Judgement must be used, cyclomatic complexity just flags *potential* issues.

Data Complexity

(Chapin 2000)

- Concerned with data complexity - based on a correlation between number of variables and complexity.
- The fewer variables you use the less complex your code.
  - Does this always hold?
  - e.g.: using the quadratic root equation every time instead of creating a new variable?
- Attempts to achieve a favourable value can greatly impact understandability/readability/modifiability.

$$CM = P + 2M + 3C + 0.5T$$

$$P = \text{Number of input variables (including globals)}$$

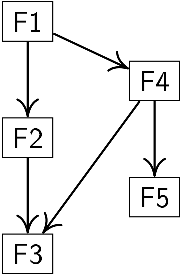
$$M = \text{Number of modified/creaded variables}$$

$$C = \text{Number of variables participating in control flow}$$

$$T = \text{Number of unused variables}$$

Fan-In & Fan-Out

- Fan-in: The number of *different* submodules that call a given submodule.
- Fan-out: the number of *different* submodules called by a given submodule.



Submodule	Fan-In	Fan-Out
F1	0	2
F2	1	1
F3	2	0
F4	1	2
F5	1	1

Fan-In & Fan-Out

Interpretation

- Fan-in: Indicates code reuse.
- Fan-out: Indicates modularity (and coupling).
- Together they indicate the structure of the code.
    - e.g.: if the *mode* of both is 1?
- Can also be applied to classes as a measure of coupling.

Coupling Between Object Classes (CBO)

Chidamber and Kemerer (1994)

- Based on Fan-out
- Often measured by a ratio (within a package/name space/ ...)

$$CBO = \frac{Number\ of\ Links}{Total\ Number\ of\ Classes}$$

- Measures the *breadth* of coupling.
- Does not consider:
  - Types of links between classes e.g.: method calls vs global variables.
  - Number of links, each *new* class is counted once and only once.

Coupling

Dhama (1995)

- CBO is appealing in its simplicity.
- Dhama calculates the coupling for each module(m_c) as a value in the range (0–1).
- Accounts for the strength of the coupling.
  - Control vs. data coupling.
  - Global variables.
  - Fan-in/fan-out
- The lower the value of m_c, the lower the overall module coupling

Coupling — Dhama1995

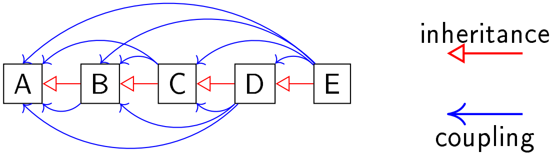
- Data and control/flow coupling:
  - d_i = number of input data parameters.
  - c_i = number of input control parameters.
  - d_o = number of output data parameters.
  - c_o = number of output control parameters.
- Global coupling:
  - g_d = number of global variables used as data.
  - g_c = number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

$$m_c = 1 - \frac{1}{d_i + (ac_i) + d_o + (bc_o) + g_d + (cg_c) + w + r}$$

Typically $a = b = c = 2$. Values can be adjusted if required.

Inheritance and Coupling

- Inheritance/polymorphism is usually used to reduce coupling.
- Inheritance itself is a form of coupling though.
  - Inheritance can easily be misused.
  - e.g.: overwriting every superclass method.
- A sub-class is coupled to *all* it's ancestors.
  - What about Java/C# Object class?
  - What about Interfaces?

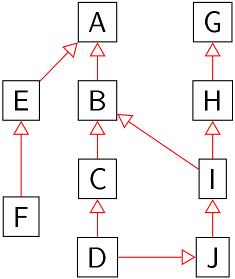


Depth of Inheritance Tree (DIT)

Chidamber and Kemerer (1994)

- DIT is the *longest* inheritance pathway (even in multiple inheritance).
  - Only measures the chain length, *not* complexity
- Can also count Number of Children (NOC).
  - Minimal use.
  - Exceedingly high number could be worth investigating.

DIT=5



Polymorphism Factor (PF)

Harrison *et al.* (1998)

- Measure coupling by method overriding.
- Calculated for each inheritance chain.
- Rule of thumb $\sim 10\%$.
- More relevant to larger systems.
- Can impact understandability.

$$PF = \frac{Number\ of\ Overrides}{Number\ of\ Opportunities} = \frac{\sum_{y \in C} M_{overrides}(y)}{\sum_{y \in C} \left(M_{new}(y) \times C_{descendants}(y) \right)}$$

Polymorphism Factor (PF)

Example

```
public class ClassA{
   public void a() {...}
   public void b() {...}
   public void c() {...}
}
public class ClassB extends ClassA{
    @Override
    public void a() {...}
}
public class ClassC extends ClassB{
    @Override
    public void c() {...}
}
```

$$Overides = 2$$

$$Oppportunities = 6$$

$$\begin{aligned} PF &= \frac{2}{6} \\ &= 0.33 \end{aligned}$$

Polymorphism Factor (PF)

Example 2

```
public class ClassA{
    public void a() {...}
    public void b() {...}
    public void c() {...}
}
public class ClassB extends ClassA{
    @Override
    public void a() {...}

    public void d() {...}
    public void e() {...}
}
public class ClassC extends ClassB{
    @Override
    public void c() {...}
}
```

$$Ov. = 2$$

$$\begin{aligned} Op. &= (3 \times 2) + (2 \times 1) \\ &= 8 \end{aligned}$$

$$\begin{aligned} PF &= \frac{2}{8} \\ &= 0.25 \end{aligned}$$

Lack of Cohesion in Methods (LCOM)

Chidamber and Kemerer (1994)

- There are 4 versions, the 4th is the most useful.
  - Earlier versions were proposed and discounted for various reasons.
- LCOM4: how many connected components are there in the class.
  - Draw a node graph.
  - Each element (method, variable, constant, etc ...) is a node.
  - Add a connection each time one element refers to another.
  - How many disjoint sets are there?
  - A value >1 indicates the class should be refactored.

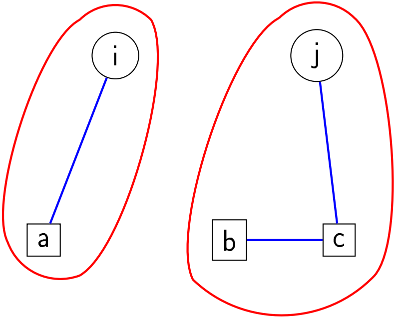
Lack of Cohesion in Methods 4 (LCOM4)

```
public class LCOM4{
    private int i;
    private int j;

    public void a(){
        i=42
    }

    public void b(){
        c();
    }

    public void c(){
        j++
    }
}
```



Tight and Loose Class Cohesion

Bieman and Kang (1995)

- Measures the *degree* of cohesion (LCOM4 is yes/no).
- Connections are based on class fields.
  - Connect methods if they share fields. Do not include fields as nodes.
- Tight Class Cohesion (TCC)
  - Directly connected methods.
  - Accounts for the density of the connections.
  - *How* cohesive is the class, for TCC=1 all methods would need to be directly connected.
- Loose Class Cohesion (LCC)
  - Similar to LCOM4.
  - The more divided the class the lower LCC.
    - i.e.: A single unconnected method gives a higher value than an equal split.

Tight and Loose Class Cohesion

Calculations

Where M is the number of methods:

- Number of Possible Connections (NP).

$$NP = M \times (M - 1)/2$$

- Direct Connections (NDC) — see Bieman and Kang (1995) for complete set theory.

$$NDC = \text{Count the lines in the graph}$$

- Tight Class Cohesion (TCC)

$$TCC = NDC/NP$$

- Indirect Connections (NIC) — as before.

$$NDC + NIC = \text{For each connected component calculate } NP$$

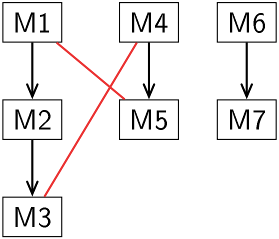
- Loose Class Cohesion (LCC)

$$LCC = NDC + NIC/NP$$

Tight and Loose Class Cohesion

Example

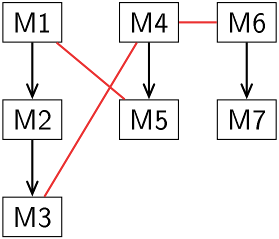
$$
\begin{aligned}
NP &= 7 \times 6/2 \\
&= 21 \\
NDC &= 6 \\
\textbf{TCC} &= 6/21 \\
&= 0.29 \\
NIC(+NDC) &= (5 \times 4/2) + (2 \times 1/2) \\
&= (10) + (1) \\
\textbf{LCC} &= 11/21 \\
&= 0.54
\end{aligned}
$$



Tight and Loose Class Cohesion

Example 2

$$
\begin{aligned}
NP =& 7 \times 6/2 \\
=& 21 \\
NDC =& 7 \\
\textbf{TCC} =& 7/21 \\
=& 0.33 \\
NIC(+NDC) =& (7 \times 6/2) \\
=& 21 \\
\textbf{LCC} =& 21/21 \\
=& 1
\end{aligned}
$$



References

Bieman, J. M. and Kang, B.-K. (1995). Cohesion and reuse in an object-oriented system. *SIGSOFT Softw. Eng. Notes*, **20**(SI), 259–262.

Chidamber, S. R. and Kemerer, C. F. (1994). A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, **20**(6), 476–493.

Dhama, H. (1995). Quantitative models of cohesion and coupling in software. *Journal of Systems and Software*, **29**(1), 65 – 74.

Harrison, R., Counsell, S. J., and Nithi, R. V. (1998). An evaluation of the mood set of object-oriented software metrics. *IEEE Transactions on Software Engineering*, **24**(6), 491–496.

McCabe, T. J. (1976). A complexity measure. *IEEE Transactions on Software Engineering*, **SE-2**(4), 308–320.