Software Metrics

Object Oriented Software Engineering
Lecture 3

Dr. Graham McDonald
graham.mcdonald@glasgow.ac.uk

Outline

- Motivation and how the quality of software can be measured
- Control Flow Graphs (CFG)
- McCabe's Cyclomatic Complexity Metrix
- CK Metrics

Desirable Properties of Software

Generally, when building software we want to:

- Reduce complexity
- Increase modularity
- Increase maintainability
  - Increase cohesion
  - Reduce coupling
- Increase reusability
- Increase usability

Motivation for Metrics

- Estimate the cost & schedule of future projects
- Evaluate the productivity impacts of new tools and techniques
- Establish productivity trends over time
- Improve software quality
- Forecast future staffing needs
- Anticipate and reduce future maintenance needs

Definitions

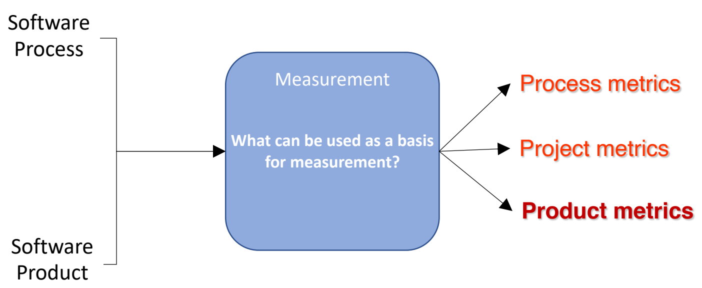
- *Measure* - quantitative indication of extent, amount, dimension, capacity, or size of some attribute of a product or process.
- *Metric* - quantitative measure of degree to which a system, component or process possesses a given attribute.
- *Indicator* - combination of metrics that provide insight into the software process or project or product itself.

How can Quality be Measured?

- To define what can be used as a basis for measurement, Bassili proposed a top-down goal oriented framework for software metrics:
  - Step 1. Develop a set of Goals
  - Step 2. Develop a set of questions that characterise the goals
  - Step 3. Specify the Metrics needed to answer the questions
  - Step 4. Develop Mechanisms for data Collection and Analysis
  - Step 5. Collect Validate and Analyse the Data.
  - Step 6. Analyse in a Post Mortem Fashion
  - Step 7. Provide Feedback to Stakeholders

Basili, Victor R. *Software modeling and measurement: the Goal/Question/Metric paradigm.* 1992.

How can Quality be Measured?



Control flow graphs (CFG) are foundational for product metrics.

Software Metrics

Object Oriented Software Engineering
Lecture 3: Part 2

Dr. Graham McDonald
graham.mcdonald@glasgow.ac.uk

Control Flow Graph (CFG)

- A representation, using **graph** notation, of all paths that might be traversed through a program during its execution.
- Nodes are basic blocks in a program code
- Edges represent possible flow of control from the end of one block to the beginning of the other
- There may be multiple incoming/outgoing edges for each block

How to draw CFG

1. Identify methods in a class

```
public void collisionDetector() {
    dts = computeDTS();
    dfo = computeDFP();
    breaking = false;
    airbagactive = false;
    alarmon = false;

    while(accelerating) {
        dts = computeDTS();
        dfo = computeDFP();

        if(dts < 10) {
            alarmon = true;
        }

        if(dts == dfo) {
            airbagactive = true;
        }
        else {
            airbagactive =false;
        }
    }

    printstatus();

}
```

How to draw CFG

2. Divide the intermediate code of each method into basic blocks. A basic block is a piece of straight line code, i.e. there are no jumps in or out of the middle of a block.

```
public void collisionDetector() {
    dts = computeDTS();
    dfo = computeDFP();
    breaking = false;
    airbagactive = false;
    alarmon = false;

    while(accelerating) {
        dts = computeDTS();
        dfo = computeDFP();

        if(dts < 10) {
            alarmon = true;
        }

        if(dts == dfo) {
            airbagactive = true;
        }
        else {
            airbagactive =false;
        }
    }

    printstatus();

}
```

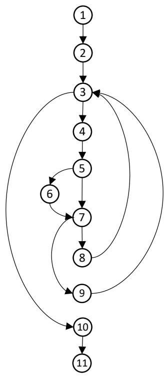
How to draw CFG

3. Add labels to the following:
   - start of the method
   - each block
   - decision points
   - end of the method

```
1  public void collisionDetector() {
       dts = computeDTS();
       dfo = computeDFP();
2      breaking = false;
       airbagactive = false;
       alarmon = false;

3      while(accelerating) {
4          dts = computeDTS();
           dfo = computeDFP();

5          if(dts < 10) {
6              alarmon = true;
           }

7          if(dts == dfo) {
8              airbagactive = true;
           }
           else {
9              airbagactive =false;
           }
       }

10     printstatus();

11 }
```

How to draw CFG

4. Generate a control flow graph of how to move from the first node to the last node

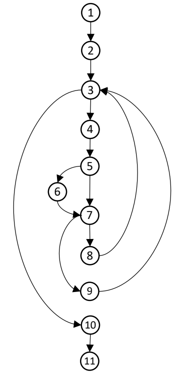



```
①  public void collisionDetector() {
       dts = computeDTS();
       dfo = computeDFP();
②      breaking = false;
       airbagactive = false;
       alarmon = false;

③      while(accelerating) {
④          dts = computeDTS();
           dfo = computeDFP();

⑤          if(dts < 10) {
⑥              alarmon = true;
           }

⑦          if(dts == dfo) {
⑧              airbagactive = true;
           }
           else {
⑨              airbagactive =false;
           }
       }

⑩      printstatus();

⑪  }
```

How to draw CFG

- An edge from one node to another node exists if execution of the statement representing the first node can result in transfer of control to the other node.

- The graph is complete if there is a path from every other node to the last node



How to draw CFG

Exercise: Draw the control flow graph for

1. An if statement.
2. A case statement.
3. A while statement.
4. A for loop

McCabe's Cyclomatic Metric

Given a control flow graph G, where the cyclomatic complexity is represented by $V(G)$, then:

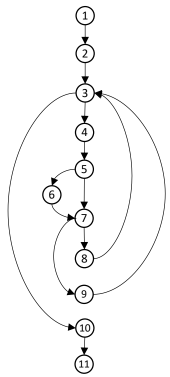
$$V(G) = E - N + 2$$

N is the number of nodes in G

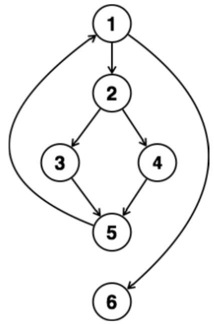
E is the number of edges in G

McCabe’s Cyclomatic Metric

Cyclomatic complexity
V(G) = 13 – 11+2 = 4



McCabe’s Cyclomatic Metric



What is Cyclomatic Complexity
of the CFG?

McCabe’s Cyclomatic Metric

- rule of thumb:
  - begin restructuring your code with the component with highest V(G)

V(G)	**Risk**
1 – 10	easy program, low risk
11 – 20	complex program, tolerable risk
21 – 50	complex program, high risk
>50	impossible to test, extremely high risk

McCabe’s Cyclomatic Metric

- Advantages
  - easy to compute (parser)
  - empirical studies: good correlation between cyclomatic complexity and understandability
- Disdvantages
  - only control flow
  - no data flow
  - may be inappropriate for OO programs (trivial functions)

Software Metrics

Object Oriented Software Engineering
Lecture 3: Part 3

Dr. Graham McDonald
graham.mcdonald@glasgow.ac.uk

CK Metrics

- In 1994, Shyam Chidamber and Chris Kemerer defined six simple metrics for object-oriented programs.
    - Since then this work has been extended to over 300 metrics.

S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," in *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476-493, Jun 1994.

CK Metrics: Objectives

- **WMC** Weighted Methods Per Class
- **DIT** Depth of Inheritance Tree
- **NOC** Number of Children
- **CBO** Coupling between Object Classes
- **RFC** Response for a Class
- **LCOM** Lack of Cohesion of Methods

S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," in *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476-493, Jun 1994.

Weighted methods per class (WMC)

- This is the sum of the complexities of methods in a class

$$WMC = \sum_{i=1}^{n} c_i$$

- c_i is the *complexity* of each method M_i of the class
- Complexity is the McCabe complexity of the method
- Smaller values are better
- WMC is a predictor of how much TIME and EFFORT is required to develop and to maintain the class.
- The objective is to achieve **low** WMC

Depth of inheritance tree (DIT)

- DIT is the length of the path from the node to the root of the tree

The greater values of DIT :

a) The greater the number of methods (NOM) it is likely to inherit, making more **COMPLEX** to predict its behaviour

b) The greater the potential **RE-USE** of inherited methods

c) Small values of DIT in most of the system's classes may be an indicator that designers are forsaking **RE-USABILITY** for simplicity of **UNDERSTANDING**.

d) The objective is to achieve the appropriate **trade-off** in DIT

Number of children (NOC)

- For any class in the inheritance tree, NOC is the number of *immediate* children of the class (the number of direct subclasses)

The greater values of NOC:

a) the greater is the **RE-USE**

b) The greater is the probability of **improper abstraction** of the parent class,

c) The greater the requirements of method's **TESTING** in that class.

Small values of NOC, may be an indicator of lack of communication between different class designers.

The objective is to achieve the appropriate **trade-off** in NOC

Coupling between Object classes (CBO)

- For a class, C, the CBO metric is the number of other classes to which the class is coupled
- A class, X, is coupled to class C if
  - X operates on (affects) C or
  - C operates on X

Small values of CBO :

- Improve **MODULARITY** and promote **ENCAPSULATION**
- Indicates independence in the class, making easier its **RE-USE**
- Makes easier to **MAINTAIN** and to **TEST** a class.
- The objective is to achieve **low** CBO

Response for class (RFC)

- It is the number of methods of the class plus the number of methods called by any of those methods.
- Normally RFC is calculated up to the first method call level, and not through the transitive closure of all method calls.
- Smaller numbers are better
  - Larger numbers indicate increased complexity and debugging difficulties. **TESTING** and **MAINTANACE** of the Class becomes more **COMPLEX**
- The objective is to achieve **low** RFC

Lack of cohesion metric (LCOM)

- Counts the sets of methods in a class that are **not** related through the sharing of some of the class's fields.
  - Step 1: Consider all pairs of a class's methods.
  - Step 2: Check if the pair share common fields.
    - In some of these pairs both methods access at least one common field of the class, while in other pairs the two methods to not share any common field accesses.
  - Step 3: The lack of cohesion in methods is then calculated by subtracting from the number of method pairs that don't share a field access the number of method pairs that do

Lack of cohesion metric (LCOM)

- A measure of the “tightness” of the code
- Greater values of LCOM:
  - Increases **COMPLEXITY**
  - Does not promotes **ENCAPSULATION** and implies classes should probably be split into two or more subclasses
- Helps to identified low-quality design
- The objective is to achieve **low** LCOM

CK Metrics: Guidelines

METRIC	GOAL	LEVEL	COMPLEXITY (To develop, to test and to maintain)	RE-USABILITY	ENCAPSULATION, MODULARITY
WMC	Low	▼	▼	▲	
DIT	Trade-off	▼	▼	▼	
		▲	▲	▲	
NOC	Trade-off	▼	▼	▼	
		▲	▲	▲	
CBO	Low	▼	▼		▲
RFC	Low	▼	▼		
LCOM	Low	▼	▼		▲