Software Design and Construction
159.251

Software Measurements and Metrics

Amjed Tahir
a.tahir@massey.ac.nz

Overview

Have you had these questions before?

- I always hear of something called "***software quality"***, but I'm not sure how someone can measure quality?

- What ***aspects of the software*** that are measurable?

- Does measuring ***software attributes*** provide any useful information?

- I have a large software development team, how to ***track the progress*** of all team members?

Software Quality Management

- Most software development projects don’t meet their success criteria:
  - Suffer from budget and schedule overrun.
- Software quality depends on several factors such as ‘on time’ delivery, within budget and fulfilling users needs.
- Quality should be maintained from the start of software development.

What makes software engineering, ***engineering***?

Measurements!

The role of measurements in Software development

"You can't control what you can't measure!"

Tom DeMarco

- Software development and implementation is centred around software measurements.

Software Measurements

- Software measurements are a quantified characteristics of software product or process…

- Measurements allow you to understand and analyse the state of your software (project and product).

- Measurements is the key to software analytics!
  - Translate subjective criteria into numbers.
  - Moving away from quantitative code reviews…

Software Measurements

cont'd

- Measurements can be applied to all stages of software development.
- From measuring requirements and specification to deployment and maintenance.
- Should be managed and controlled – for reliable measurements values (to avoid what's known as data-quality issues!)

Software Metrics

- ***Software metric*** is a quantitative measure of individual software attributes.
- Tools for anyone involved in software development to understand various aspects of the software program (code base and/or project progress).
- Metrics vs Measurements:
  - Both terms are (incorrectly) used interchangeably
    - Metrics is a function or tool
    - Measurements are the numbers obtained by the application of metrics.
- Metrics are only useful if you know what to do with the values that they provide.

When to use software metrics?

- You can use metrics at different stages of development
  - Gathering of information – to understand what’s going on!
  - Tracking progress
  - Evaluation
  - Estimation
  - Prediction – future....
  - Improvement

Benefits of metrics

- Software metrics can be very useful to use for the following:
  - ***Production planning***
  - ***Project scheduling***
  - ***Production monitoring and control***
  - ***Decision support***
  - ***Cost and benefit analysis***
  - ***Learning from experience***

Example of areas where metrics can be beneficial

- Measure developers productivity
  - How many tasks were completed this week?
- Documentation completion rate
  - Have all tasks/requirements been documented?
- Code complexity
  - How complex is each component?
- Defects and bugs density
- Performance

Different types of Software Metrics

- ***Project*** and ***Process*** metrics
  - Metrics to measure project characterises
  - Usually used by project managers and team leaders
- ***Product*** metrics
  - Focus on the characterises of the software product (written software)
  - Mostly related to the final quality

Project and Process metrics

- Can be used to measure development progress.
- Can be used for future estimation and prediction
  - Cost, effort and resources estimation
    - Through statistical analysis e.g., using regression analysis and machine learning
- A control tool?
  - Measure developers performance
  - Control resources, e.g. human resource (how many developers should work on certain tasks?)

Project and Process metrics

Cont'd

- Commonly used by management to check budget and office procedures for efficiency
- Evaluate and track aspects of the software design process like:
  - Human resources
  - Time
  - Schedule
  - Methodology

Example of *Project* metrics

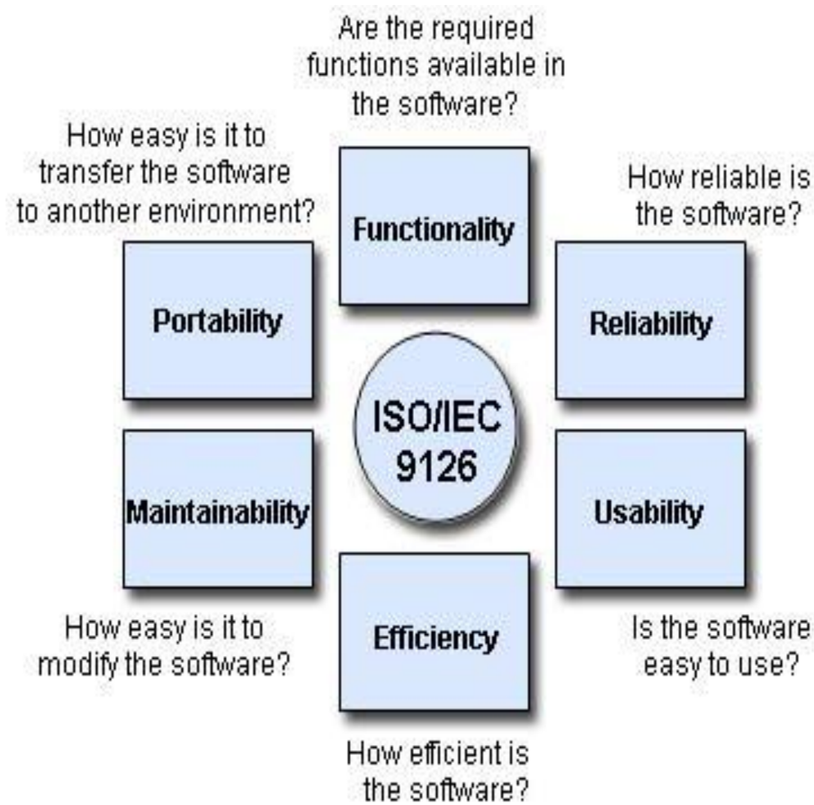
- Time taken to complete a task
- Task completions rate
- Resources required to complete a task
- Staff hours spent on life cycle activities.
- Correctness:
  - Time of finding fault/bug
  - Time to fix a fault/bug

Product Metrics

- Applied to all product characteristics.
- Design and source code metrics are widely used.
- Also can be used to measure progress and productivity.
- Strongly related to quality attributes such as maintainability and performance

Quality and Metrics

- The ISO/IEC 9126 is product quality model that defines a set of quality attributes
  - Also introduce a set of metrics to measure these attributes.
- Define 5 main attributes and a set of sub-attributes
- Metrics have been defined to measure these attributes (but mostly indirectly!)



https://www.dlsweb.rmit.edu.au/toolbox/ecommerce/gqm_respak/gqm_e1/html/gqm_e1_criteria.htm

How to measure software design?

- Measure complexity of the design.
- Widely used in Object-Oriented development.
- Can provide an early assessment of the software product
  - Even before writing any code!!
- This can be helpful for estimation and early prediction, for example:
  - ***Estimation***: of the effort needed to code.
  - ***Prediction***: of the number of faults in a piece of code.

Source code

- Measure everything in the code from complexity to confirming to coding standard.
- Even *comments* can be measured (i.e., there are metrics to measure comments ration!).
- Not all metrics are universal
  - Some metrics are specific to a programming paradigm or language (especially code-based metrics)
  - Design metrics can be similar across languages (especially if they follow the same paradigm e.g., Object oriented)

Static and Dynamic metrics

- Static metrics are the class of measures that capture the *static* properties.
  - Does not need the software to be executed
  - Does not change if you run the program

- Dynamic metrics measure the runtime prosperities
  - Need to run the program
  - Accurate metrics ➔ measures actual behaviour
  - Harder to collect and analysis (different execution scenario give you different results!)

Lines of Code

- A widely used software metrics
- Counts the number of lines of code of a method/class/package/program etc…
- Widely used as a measure of software size
  - But also used as a proxy metrics for other things such as:
    - **Productivity**: i.e., man/hour
    - **Complexity** ➔ the larger the program, the higher the complexity!
    - **Maintainability** (i.e., how easy to maintain a software?)

Different variation of LOC

- Be carful when you use LOC!
- There are several definitions for LOC.
- Take these example:
  - A small program to find if X>5 or X<5

```
x=5
if x >= 5:
    print("value is greater than 5")
else:
     print("value is smaller than 5")
```

LOC = 5

```
x=5
if x >= 5: print("value is greater than 5")
else:  print("value is smaller than 5")
```

LOC =3

  - The same program, different LOC results.

```
x=5
if x > 5:
    print("value is greater than 5")
If x ==5
 print("value equals 5")
else:
     print("value is smaller than 5")
```

LOC =7

- **brackets** dependent languages will show more variations

```
x=5
if x >= 5
    {
     print("value is greater than 5")
     }
else:
     {
      print("value is smaller than 5")
     }
```

```
x=5
if x >= 5
    {
     print("value is greater than 5")}
else:
     {
      print("value is smaller than 5")}
```

LOC = 9 OR 5?

LOC definitions

- Physical and Logical LOC
  - Can you give a definition for both?

Physical and Logical LOC

- There is no precise definition for both!
  - Physical: usually include all text within the file.
    - Some metrics exclude comments
  - Logical : executable lines of code only!
    - Always exclude comments
    - Some excludes *annotations* in the code!
    - Brackets are also excluded

LOC examples

- Example

```
x=1
# x within a loop!
While x <5
    {
#  print count
     print("Count =" + x)
     x +=1;
     }
```

What's the physical and logical LOC for the above code?

LOC examples

Cont'd

```
x=1
# x within a loop!
While x <5
    {
#  print count
      print("Count =" + x)
      x +=1;
      }
```

Physical LOC = 8 ➔ Counts everything !

Logical LOC = 4 ➔ Counts only executable code!

How useful is LOC?

- LOC may just provide an indicator of the program size.
- Works well to compare programs written in the same language.
- Using LOC for comparing different languages can be ineffective
- "Hello world" written in Python, C++ and Java

```
Print  ("hello world")
```

```
Void main ()
{
Printf ("\nHello world\n");
}
```

```
Public class HelloWorld
{
  public static void main(String[]  args)
  {
   System.out.println("Hello, World");
  }
}
```

1 LOC

2 LOC

3 LOC

Other size metrics

- Number of classes (in OO programming)
- Number of methods
- Number of files
- Number of operators
- Function points

How big are software programs?

- Program can be as small as a few lines of code.
- There is no standard to define programs' size.
- Some classification for Open Source:
  - **Tiny**: fewer than 1 KLOC (kilo LOC)
  - **Small**: 1 up to 10 KLOC
  - **Medium**: 10 up to 100 KLOC
  - **Large**: 100 up to 1000 KLOC
  - **Extra-large**: more than 1000 KLOC.

- Known programs:
  - PWB/Unix v.1.0 (1979) = 100 KLOC
  - Linux Kernel (1991) = 10 KLOC – now = 19 million LOC
  - Google Chrome = 14 million LOC
  - Mozilla FireFox = 10 million LOC
  - Android = 12 million LOC (I assume it include some of Linux Kernel code!)
  - Microsoft Office 2013 = 40 million
  - Debian 5.0= over 60 million LOC!

  Read more:

  http://www.informationisbeautiful.net/visualizations/million-lines-of-code

Cyclomatic Complexity

- It is useful to be able to quantify the complexity of the program or individual modules within the program (e.g, to predict the error-prone sections).

- Cyclomatic complexity (CC) is a metric that provides a quantitative measure of the **logical complexity** of a program module. It is recommended that the CC associated with various modules should not exceed 10.

- The value computed for Cyclomatic complexity defines ***the number of independent paths*** for a particular module that must be tested in order to ensure that ***all*** the **statements and branches have been executed** *(at least once!)* by the various test cases.

```
/* Binary search

Public static void search(int key, int[] elemArray, Result r) {
int bottom = 0;
int top = elemArray.length-1;
int mid;
r.found = false;
r.index = -1;
while (bottom <= top) {
        mid=(top+bottom)/2;
        if (elemArray[mid] == key) {
                r.index = mid;
        r.found = true;
        return;
                }
              else {
        if (elemArray[mid] < key)
              bottom = mid+1;
               else
              top = mid-1;
                 }
             }
}
```

The previous Binary search program assumes a sorted list:

Consider the following list and a search key such as “9”:

1
3
4
6
7
9
11
15

The key is compared to the middle element in the list, if the test is successful the search ends. Otherwise depending on whether the key is less than or greater than the middle element the search is continued on the top half or bottom half of the list.

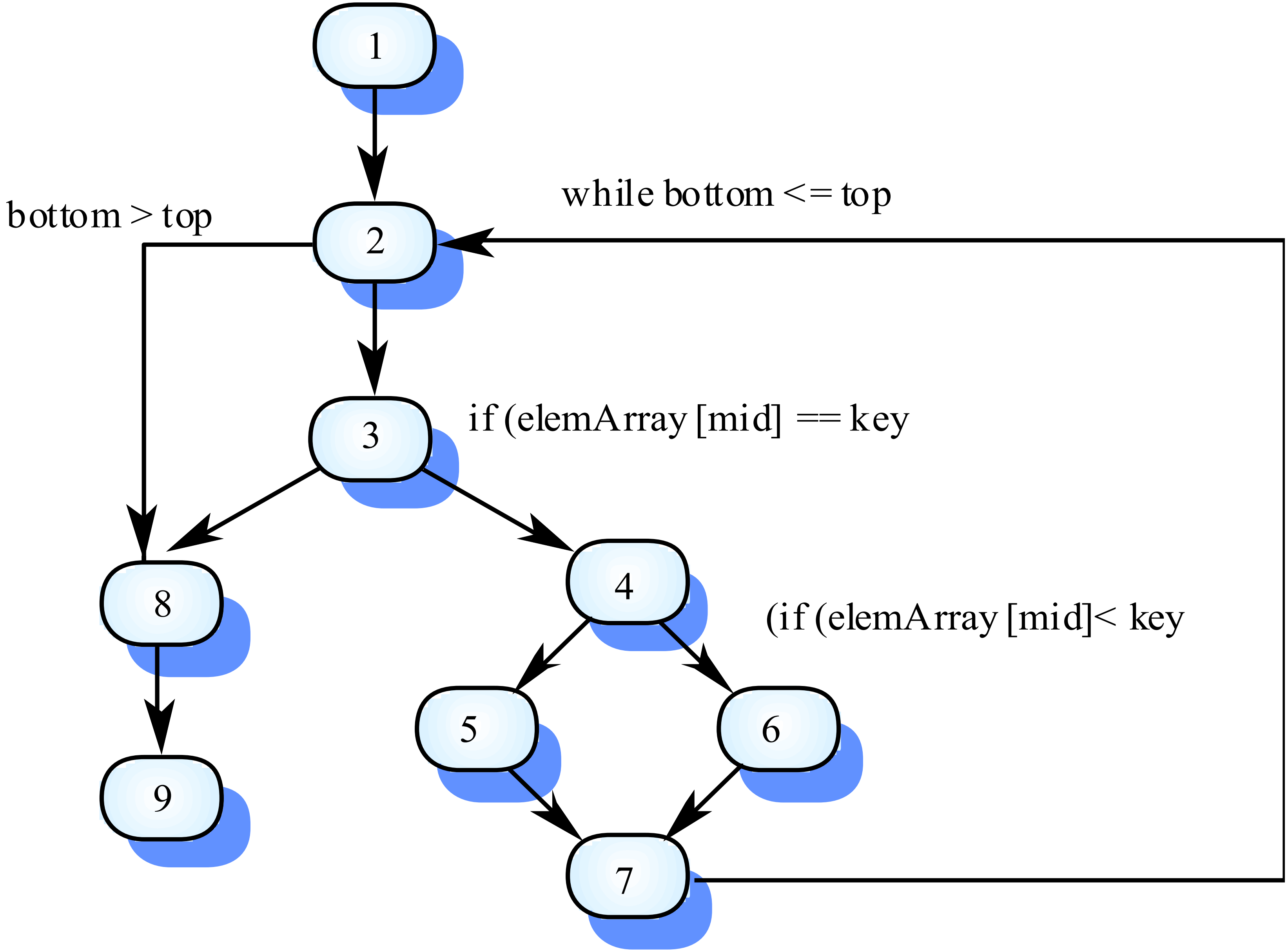
```
Public static void search(int key, int[] elemArray, Result r) {
int bottom = 0;
int top = elemArray.length-1;
int mid;
r.found = false;
r.index = -1;
while (bottom <= top) {
            mid=(top+bottom)/2;
            if (elemArray[mid] == key) {
             r.index = mid;
            r.found = true;
            return;
             }
           else {
            if (elemArray[mid] < key)
            bottom = mid+1;
             else
                 top = mid-1;
               }
         }
}
```

CC = # of branching conditions + 1

```
Public static void search(int key, int[] elemArray, Result r) {
int bottom = 0;
int top = elemArray.length-1;
int mid;
r.found = false;
r.index = -1;
while (bottom <= top) {
        mid=(top+bottom)/2;
        if (elemArray[mid] == key) {
        r.index = mid;
        r.found = true;
        return;
        }
       else {
        if (elemArray[mid] < key)
        bottom = mid+1;
         else
            top = mid-1;
          }
      }
}
```

CC = # of branching conditions + 1

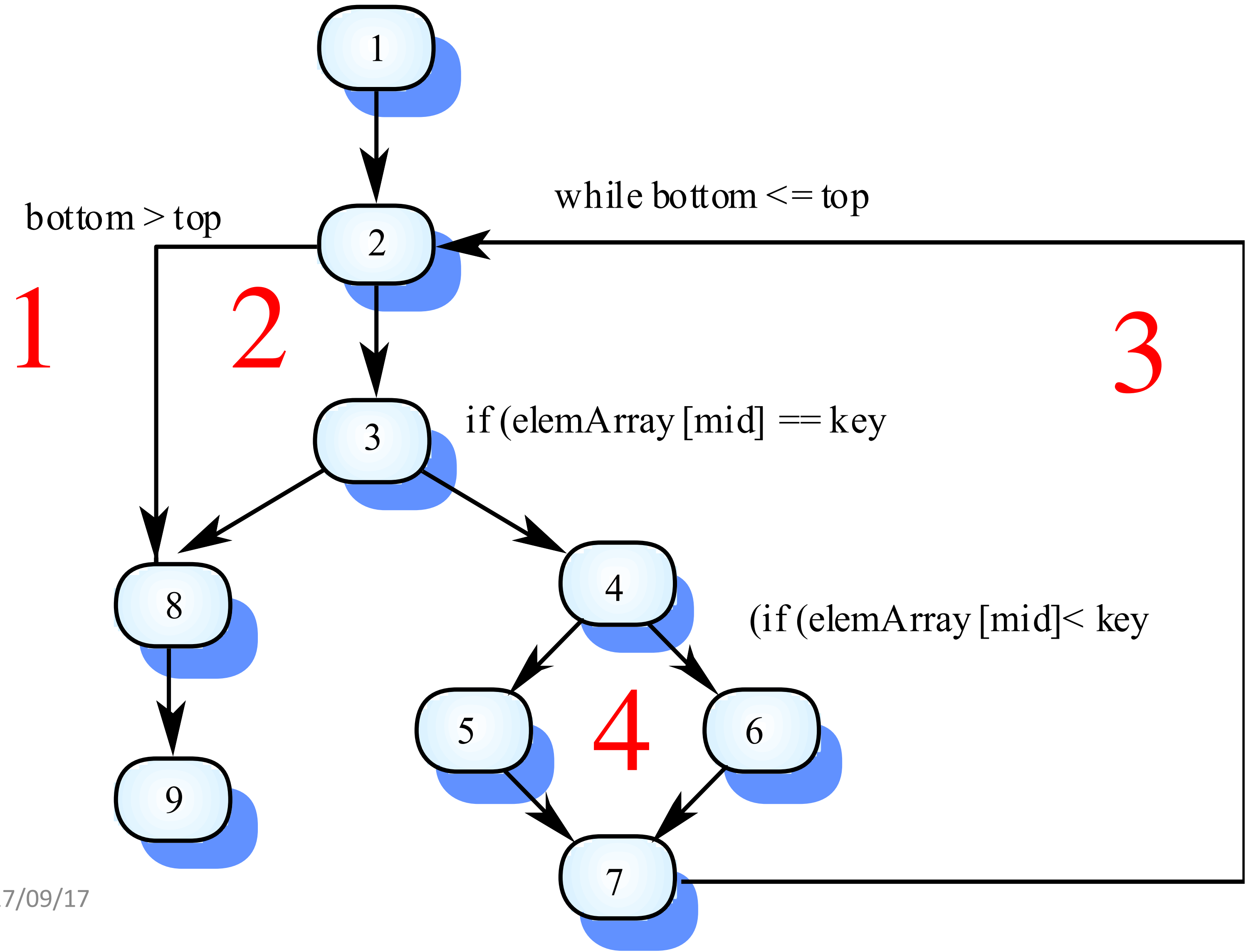
Cyclomatic complexity=3+1=4



Binary search flow graph

Calculating cyclomatic complexity (another method)

The cyclomatic complexity = the number of regions in the graph including the universal region (the outer region).



Calculating cyclomatic complexity

- The cyclomatic complexity, CC of a graph, with **N vertices** and **E edges** is

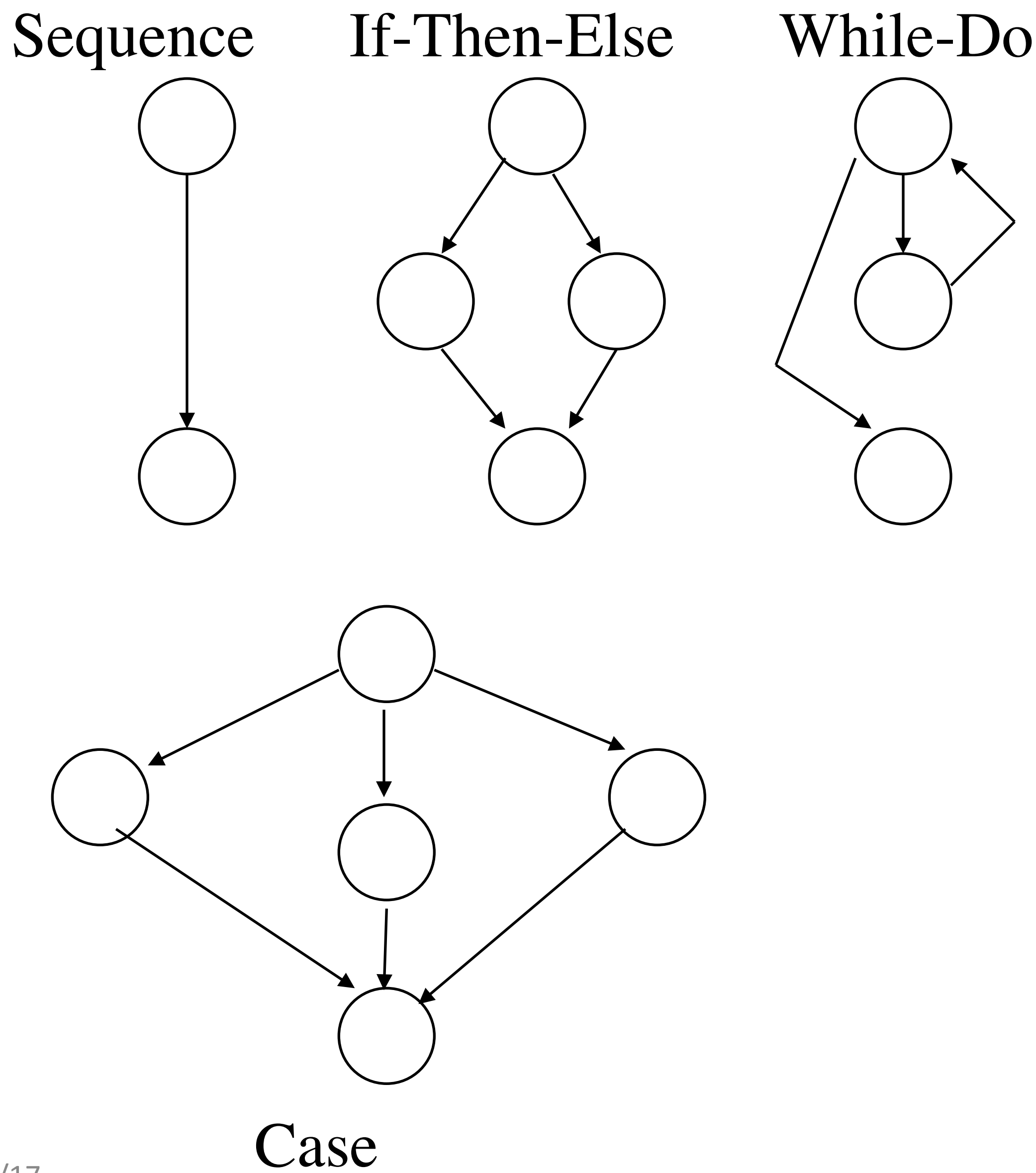
  CC= E - N + 2

  The cyclomatic complexity for Binary Search

  = 11 – 9 + 2

  = 4

Basic Control Constructs



Inheritance and Testability

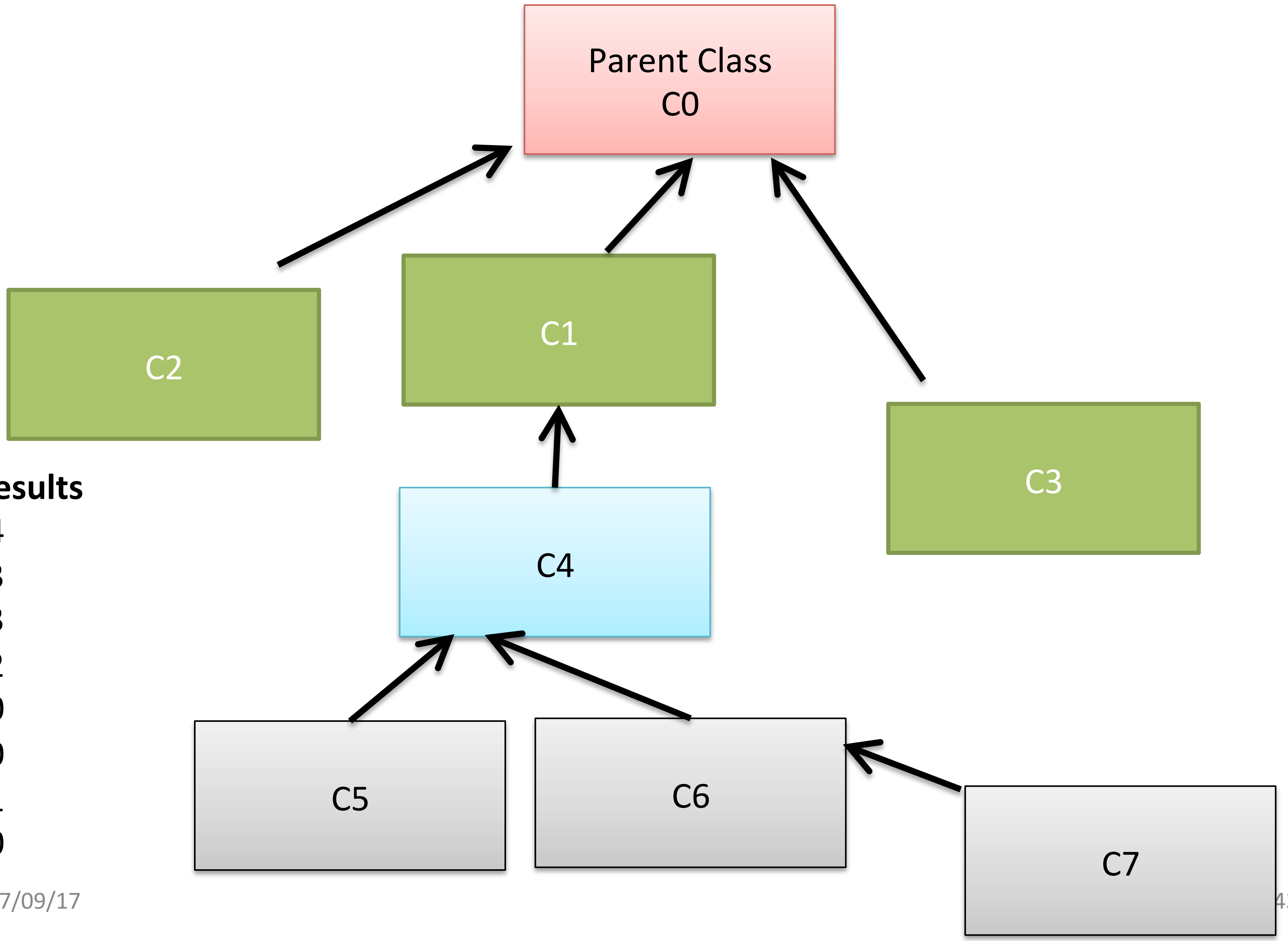
- **What's Inheritance?**

  A class (or object) is able to inherit characteristics from another class.

  A Parent (super)- Child (sub) relationship.

- **How to measure Inheritance?**
  - *<u>Depth of Inheritance tree (DIT)</u>*
  - Example is in the following slide.
- **How inheritance affects quakity?**
  - Classes that are deep down in the hierarchy potentially inherit many methods from super-classes.
  - Good coverage requires more testing paths
  - Execution paths increases with number of inherited classes

Problems with Inheritance

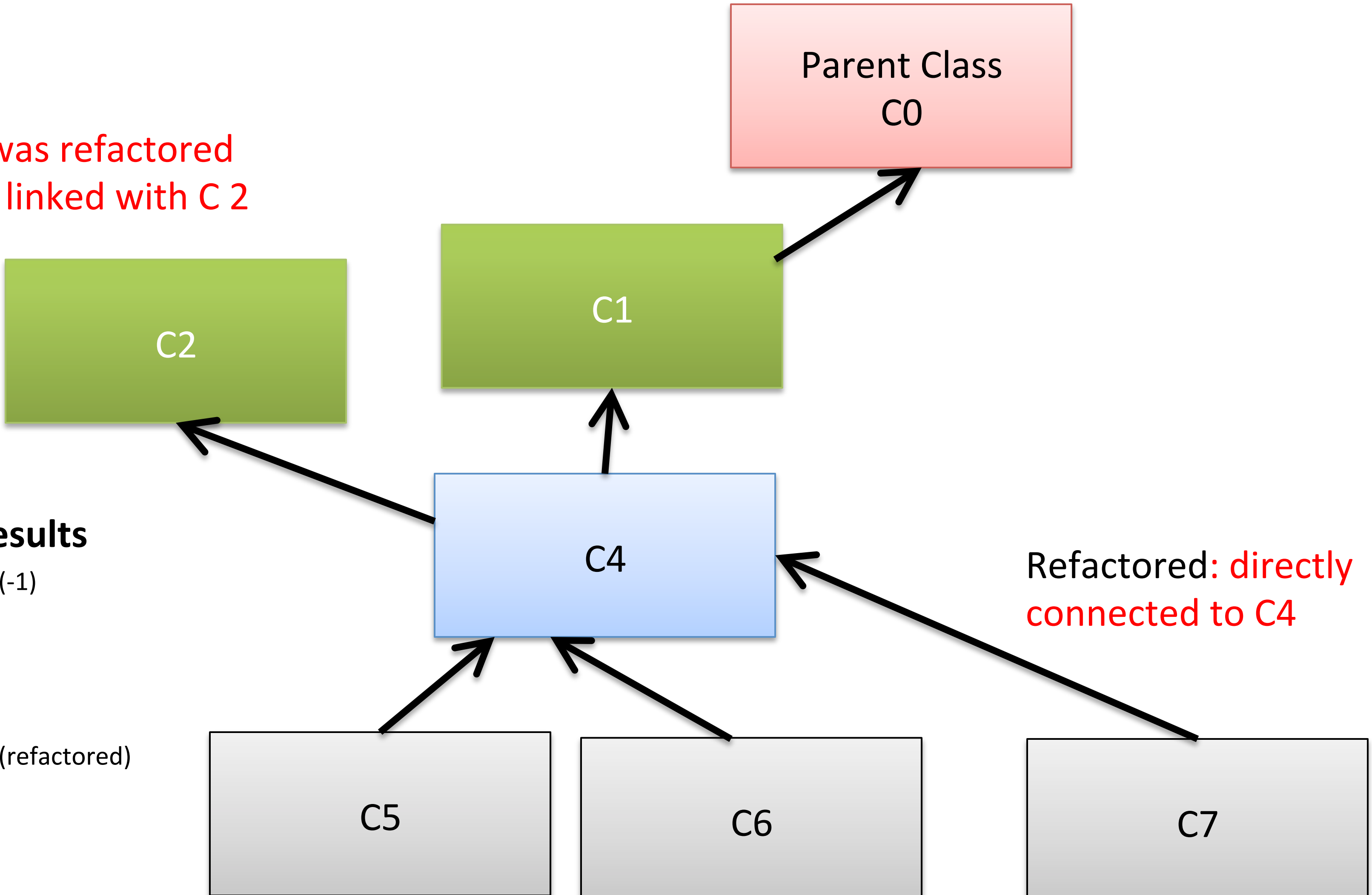


DIT Results

C7 = 4
C6 = 3
C5 = 3
C4 = 2
C3 = 0
C2 = 0
C1 = 1
C0 = 0

Inheritance: After refactoring

C3 was refactored
and linked with C 2



Refactored: directly
connected to C4

DIT Results
C7 = 3 (-1)
C6 = 3
C5 = 3
C4 = 2
C3 = 0 (refactored)
C2 = 0
C1 = 1
C0 = 0

Coupling and Quality

- **What's Coupling?**
  - Two class are coupled if at least one of them acts upon the other
    - Example: a method declared in one class uses another method(s) or instance variable(s) defined by another class
- **How to Measure Coupling ?**
  - Coupling Between Objects (CBO)
  - Import and Export Coupling
- **How Coupling Affects Testability?**
  - Tightly coupled classes are highly dependable on other classes
  - Change in one may require change in all related classes.
  - Good coverage requires more testing paths (less dependencies!)

```
public class foo {

public Method_foo()
   {
     Method_ bar1()
}
}
```

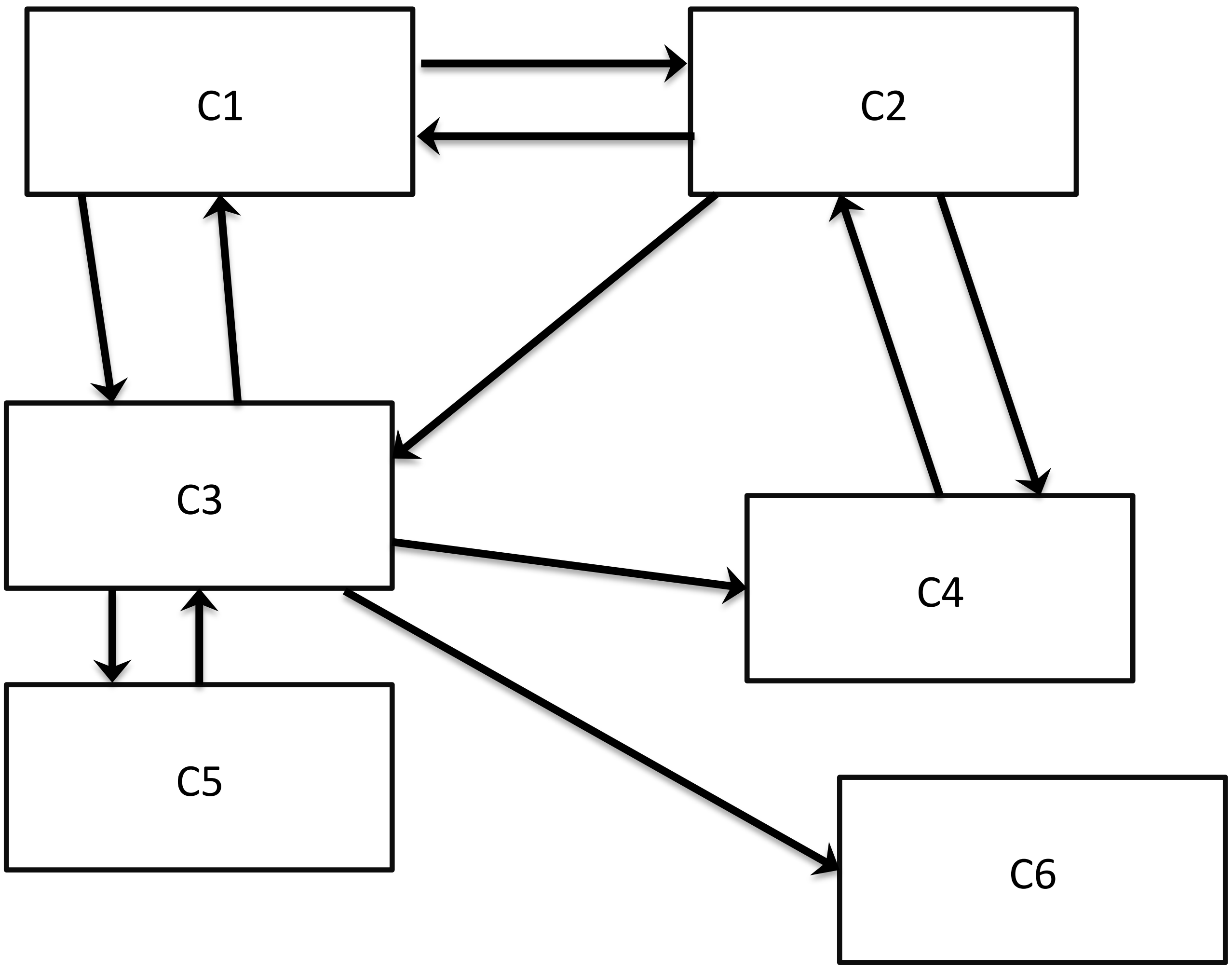
```
public class bar{

public Method_bar1()
   {
     print (something)
}

public Method_bar2()
   {
   foo()
}


}
```

Coupling : Example 1

EC Results (sent)
C6 = 0
C5 = 1
C4 = 1
C3 = 4
C2 = 3
C1 = 2

IC Results (Received)
C6 = 1
C5 = 1
C4 = 2
C3 = 3
C2 = 2
C1 = 2

Average Coupling
C6 = ((0 + 1) / 22)*100 = 5%
C5 = ((1 + 1) / 22)*100 = 9%
C4 = ((1 + 2) / 22)*100 = 14%
C3 = ((4 + 3) / 22)*100 = 32%
C2 = ((3 + 2) / 22)*100 = 23%
C1 = ((2 + 2) / 22)*100 = 18%



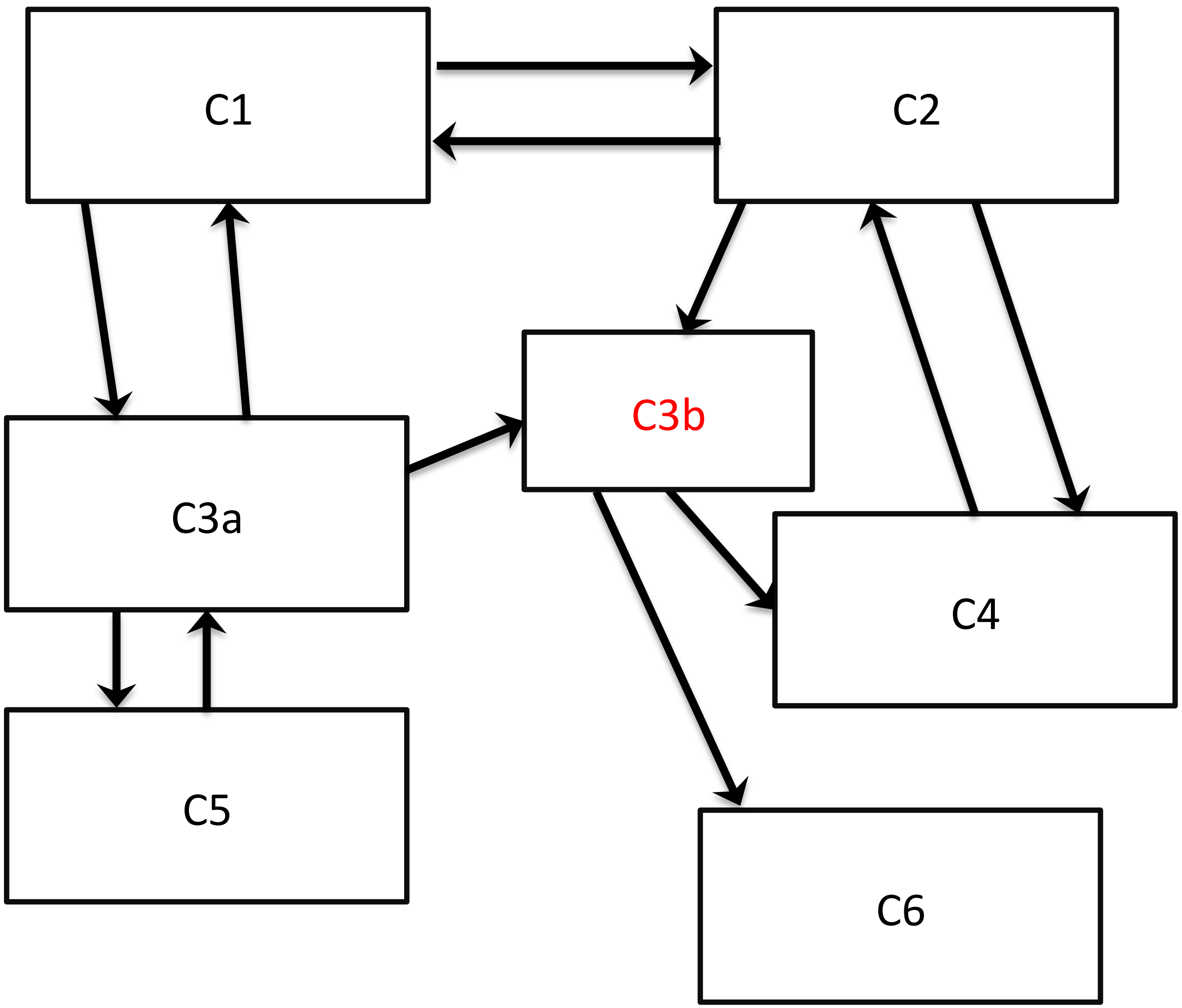
Coupling :

Refactoring 1 (add class)

EC Results (sent)
C6 = 0
C5 = 1
C4 = 1
C3a = 3
C3b = 2
C2 = 3
C1 = 2

IC Results (Received)
C6 = 1
C5 = 1
C4 = 2
C3a = 2
C3b = 2
C2 = 2
C1 = 2

Average Coupling
C6 = ((0 + 1) / 24)*100 = 5%
C5 = ((1 + 1) / 24)*100 = 8%
C4 = ((1 + 2) / 24)*100 = 13%
C3a = ((3 + 2) / 24)*100 = 21%
C3b = ((2 + 2) / 24)*100 = 17%
C2 = ((3 + 2) / 24)*100 = 21%
C1 = ((2 + 2) / 24)*100 = 17%



Coupling:

Refactoring 2 (marge class)

EC Results (sent)

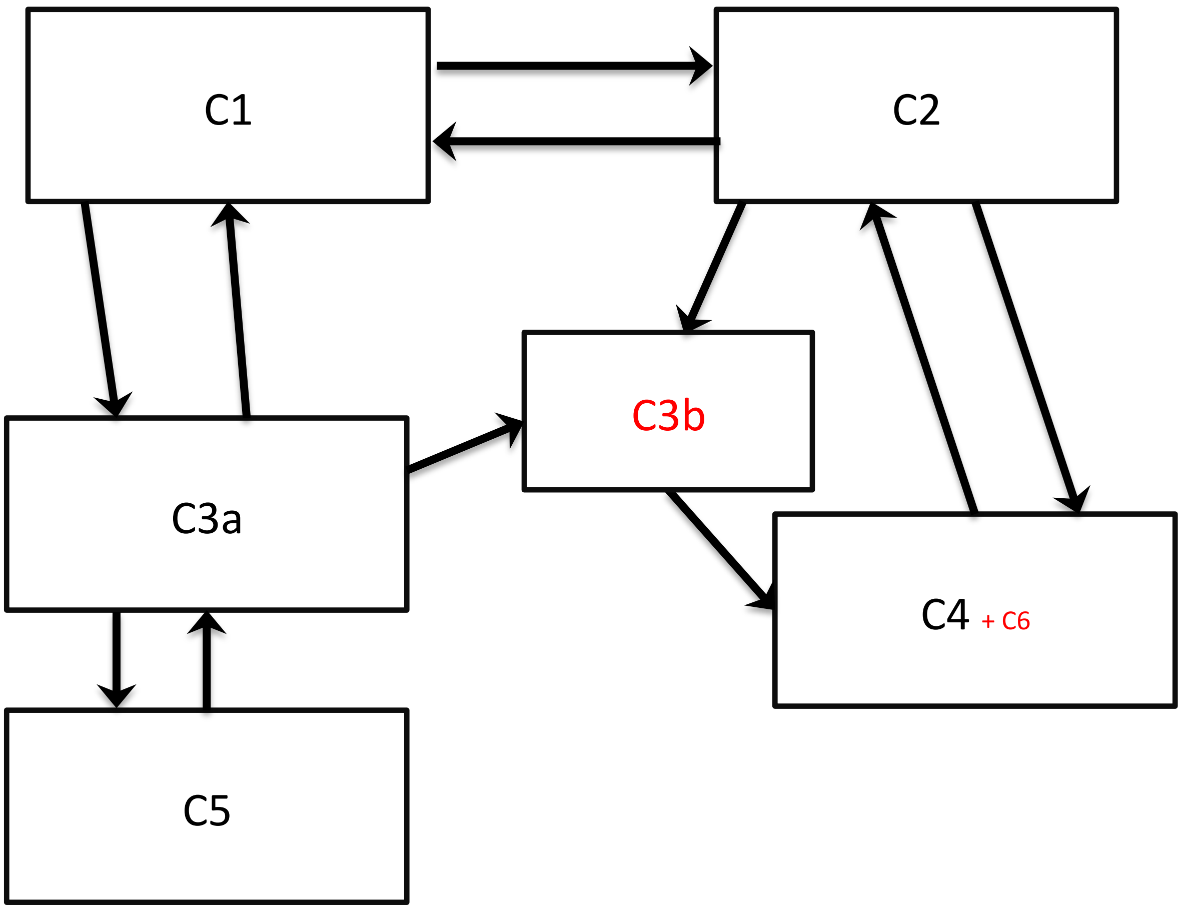
C5 = 1
C4 = 1
C3a = 3
C3b = 1
C2 = 3
C1 = 2

IC Results (Received)

C5 = 1
C4 = 2
C3a = 2
C3b = 2
C2 = 2
C1 = 2

Average Coupling

C5 = ((1 + 1) / 22)*100 = 9%
C4 = ((1 + 2) / 22)*100 = 14%
C3a = ((3 + 2) / 22)*100 = 23%
C3b = ((1 + 2`) / 22)*100 = 14%
C2 = ((3 + 2) / 22)*100 = 23%
C1 = ((2 + 2) / 22)*100 = 18%



Metrics and Faults....

- Several metrics are used to measure faults..
  – There is an assumption that the number of faults in a program increases with the size and the complexity of the program.
- Example

  -faults fixing rate,
- Some product metrics can be used to predict faults.
  – Including : Coupling metrics, Inheritance metrics, Dependency metrics etc…

Metrics from your VCS

- Many metrics from the version control system can be used to measure progress and predict changes.
  - Measure productivity:
    - Number of commits, number of changes, etc…
  - Measure complexity
    - Size of commit, number of merges
    - Number of authors worked on a file…
- Recent findings shows such metrics to be very effective in detecting faults/bugs!
  - I.e., more authors/commits/changes = more bugs

Other important metrics

- **Documentation metrics:**
  - Readability Index
  - Comments percentage
- **Resource metrics:**
  - Staff/hours spend on life cycle activities.
  - Task completion rate per staff

- **Requirements metrics**
  - Size of requirements:
    - looks into how big is each requirements - helps in planning and resource allocation.
  - Traceability
    - Tracing the successful implantation of requirements
  - Completeness
    - how many requirements have been successfully implemented (i.e., completion ratio)

- **Test metrics**
  - Fault density
  - Fault injection ➔ mutation testing
  - Defect Removal Efficiency (DRE)

    DRE = errors/(errors+defects)

    errors are faults found during the development, and

    defects are faults found after software release by the customer.

    the objective is to achieve a DRE > 0.95.

- **Code coverage:**
  - Part of testing metrics
  - Important metrics to assess how much of you code is covered by tests.
  - It is assumed that the higher the coverage, the better the test (i.e., more faults will be discovered).
  - Different types of coverage
    - Can you name a few?

- Example of coverage metrics
  - **Statement coverage:** how many statements (e.g., LOC) are covered by your test.
  - **Class coverage:** how many classes are covered by your test?
  - **Branch coverage:** if else!
  - **Condition coverage:** check if each Boolean expression (True False) has been evaluated.

- **Communication metrics**
  - This group concerns the communication patterns between different team members
  - Some large organization uses such metrics to locate issues and improve productivity!
    - Example: emails sent/received, meetings etc…

- **Performance metrics**
  - One of the widely used group of metrics.
  - ***Measurements*** is key for any *performance* assessment. Otherwise, how would you known that a program performed well/bad?
  - Critical-life (or mission-critical) systems depend heavily on metrics.
  - Deal with errors and recovery
  - Assess failures based on time…
    - Mean Time To Failure (MTTF) and Mean Time Between Failures (MTBF)
      - Mostly depend on hard-software dependencies.

Common Issues

- The managers/developers conflict!
  - Developers may resist using metrics to track their progress.
  - The metrics culture ➔ Who would agree to report their metrics data?
- Some metrics can be hard to collect (especially at project level).
  - Product metrics can be misleading
    - the case of LOC and cyclomatic complexity

Automated tools

- Several tools can be used to collect different set of metrics
- Most available tools are product-focused.
  - With the majority focus on source code metrics.
- Most metrics tools are language-specific.
  - E.g., focus on one or couple of similar languages
- Integrates with IDEs ➔ provides within development tracking
  - For example, many metrics tools integrates well with Eclipse and Microsoft Visual Studio.

Build your own

- Sometimes, you will need to build your own tool
- Or, customise one for your needs.
- Specific metrics needs to be collected in specific ways.

- Some metrics programs can be really expensive!

Example of metrics tools

- **PMD**
  - Collect a large set of metrics for a number of programming languages such as Java and
  - Can be installed as a plugin in a number of IDEs such as Eclipse, IntelliJ and NetBeans.
- **Sonar**
  - Similar to PMD, but with more metrics and better control.
    - https://sonarqube.com
  - Supports more than 20 languages including Java and Python.

Other useful tools…

- **Campwood Software**
  - http://www.campwoodsw.com/sourcemonitor.html

- If you work Java, you should try the **Eclipse Metrics Plugin** (one of my favourites - I use it in my research as well!!)

Useful resources

- Kaner, C and Bond, W, Software engineering metrics: What do they measure and how do we know. *In 10th International Software Metrics Symposium*. 2004.
- Maraca, s, n.d. , Software Measurements, available from http://www.uio.no/studier/emner/matnat/ifi/INF5181/h11/undervisningsmateriale/reading-materials/Lecture-06/morasca-handbook.pdf
- Software Quality metrics, http://www.sqa.net/softwarequalitymetrics.html