Software Systems

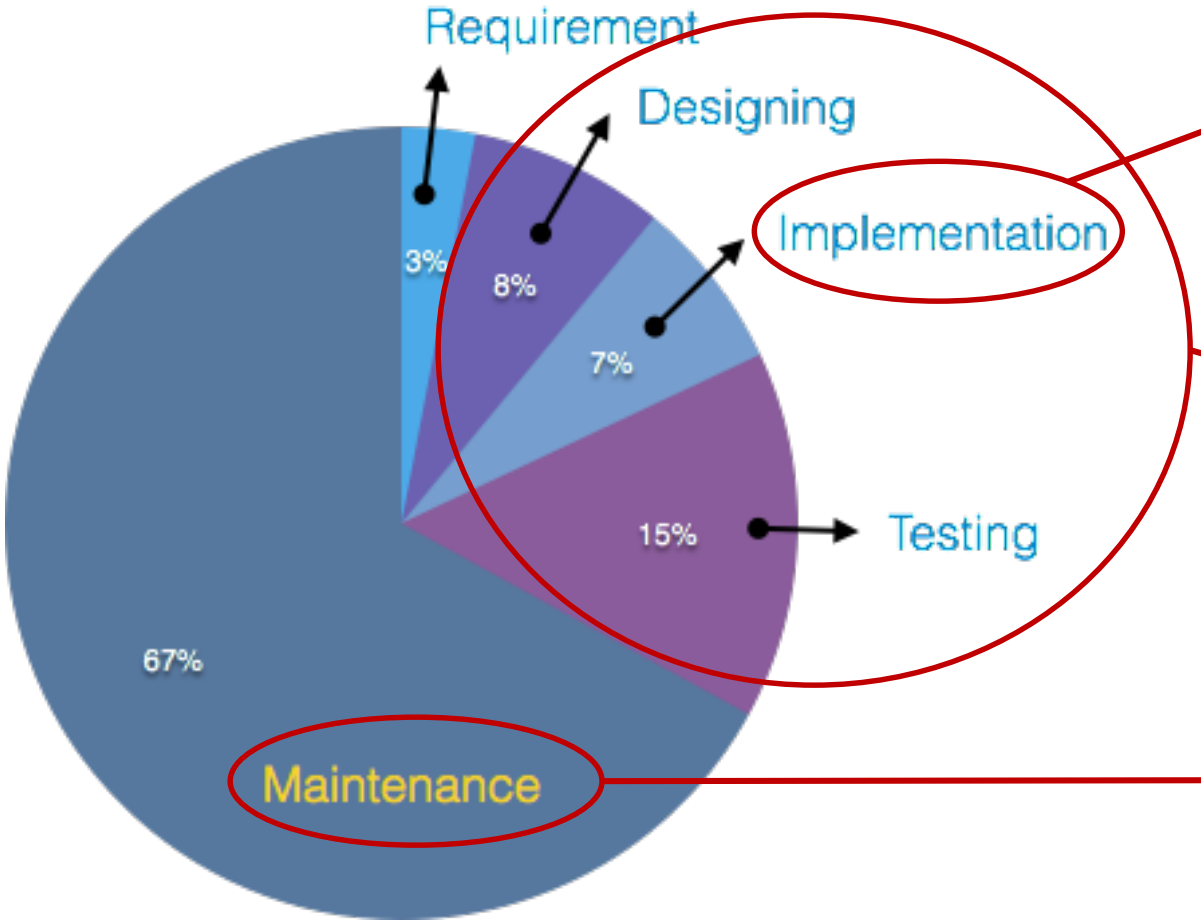
Design lecture 4 Maintainable Software and Software Metrics

Klaas Sikkel

Contents

- Brief introduction to Software Maintenance
- Some software metrics in detail
  - Size and complexity of methods
  - Coupling and cohesion of classes/packages
- Related maintenance concepts
  - Legacy
  - Technical Debt
  - “Code Smells”

Software development costs



This is what many TCS students like best

This is what you are learning about in this course

This is what programmers spend most of their working life doing

Source: http://www.tutorialspoint.com/software_engineering/software_maintenance_overview.htm

Famous historical example: “Y2K”

- Programmers had not anticipated that their software still would be running in the 21st century
  - year represented as 2 digits (e.g. ‘67’ for 1967), needed to be changed to 4 digits.

  - Estimated costs for the USA alone:
    US $ 100 billion (= $ 365 per US resident)

Source: Economics and Statistics Administration, US Department of Commerce, Nov 1999

Why do we need maintenance?

- Someone found a bug that need to be fixed
- Clients want additional features
- The system is dependent on software/ hardware which is no longer supported
- The business environment and/or the legal requirements have changed
- Two companies merge and their systems need to be integrated

Types of maintenance

- **Corrective maintenance:** Bugs have been discovered and need to be fixed
- **Adaptive maintenance:** The system has to be adapted to changes in the environment
- **Perfective maintenance:** New or changed requirements call for new features
- **Preventive maintenance:** The quality and maintainability of the software is increased, future bugs are prevented (***Refactoring***)

Software Metrics

- LOC: # Lines Of Code per method (i.e., **Size**)
- CC (Cyclomatic Complexity):
  Complexity of control flow within a method
  - WMC: Weighted Methods per Class
- CA/CE: **Coupling** between classes (packages)
- LCOM: Lack Of **Cohesion** in Methods

LOC (or MLOC): Lines of Code per Method

- Principle: larger methods are more difficult to comprehend and maintain

- Guideline for Java: generally ≤ 15 [Visser 2016]
  - What is a reasonable number may depend on the programming language used
- How do you measure LOC?
  You can tweak the lay-out…
- If you use standard lay-out conventions it is an objective measure of code size

How to interpret metrics guidelines

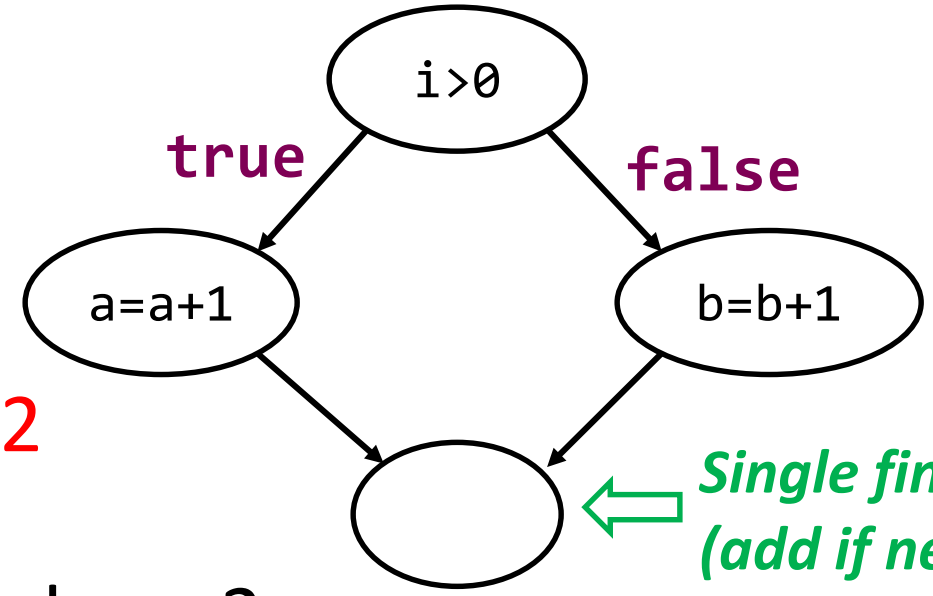
- If a few methods (classes, packages) do not comply with the guideline, that is OK if you know what you're doing
  - There can be good reasons to ignore guidelines in special cases

- *Software Improvement Group* has developed ratings for various metrics. For example:
  For LOC you get 4 stars (out of 5) if your code satisfies.
  - max 7 % of methods has LOC > 60
  - max 22 % of methods has LOC > 30
  - max 44 % of methods has LOC > 15
  - min 56 % of methods has LOC ≤ 15

Cyclomatic Complexity

- Control flow can be represented by a graph

```
if (i > 0) {
    a = a+1;
} else {
    b = b+1;
}
```



M=2

Single final node (add if needed)

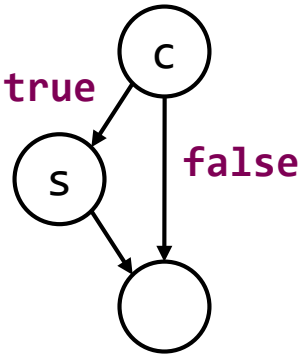
M = #edges – #nodes + 2

- Defined by Thomas McCabe (1976)
- Cyclomatic Complexity is abbreviated **CC** or **VG** or **V(G)**

Flow graphs for basic control constructs

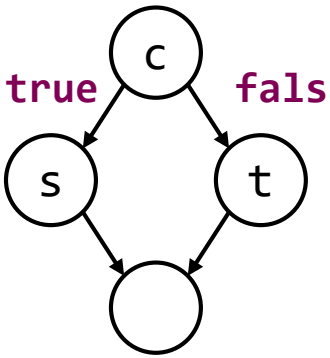
```
if (c) { s }
```

M=2




```
if (c) { s } else { t }
```

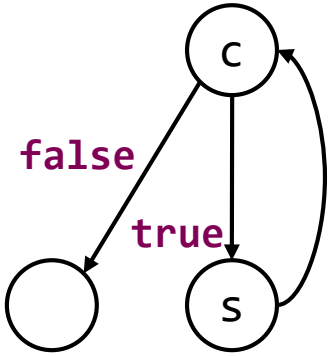
M=2




```
switch (c) {
case s1:
  break;
case s2:
  break;
…
case sn:
  break;
}
```

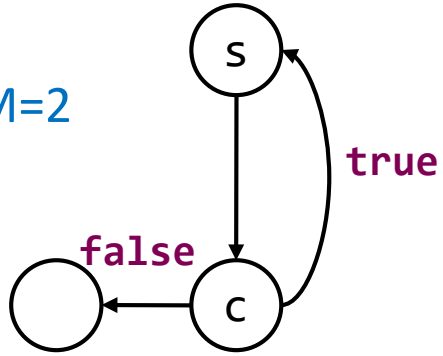
```
while (c) { s }
```

M=2

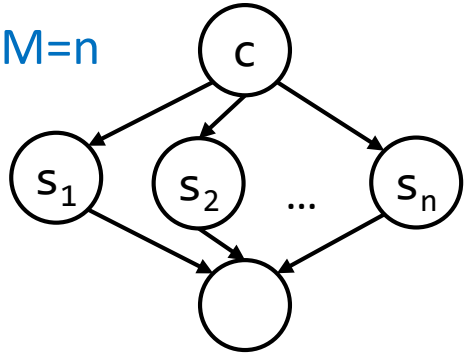



```
do { s } while (c)
```

M=2

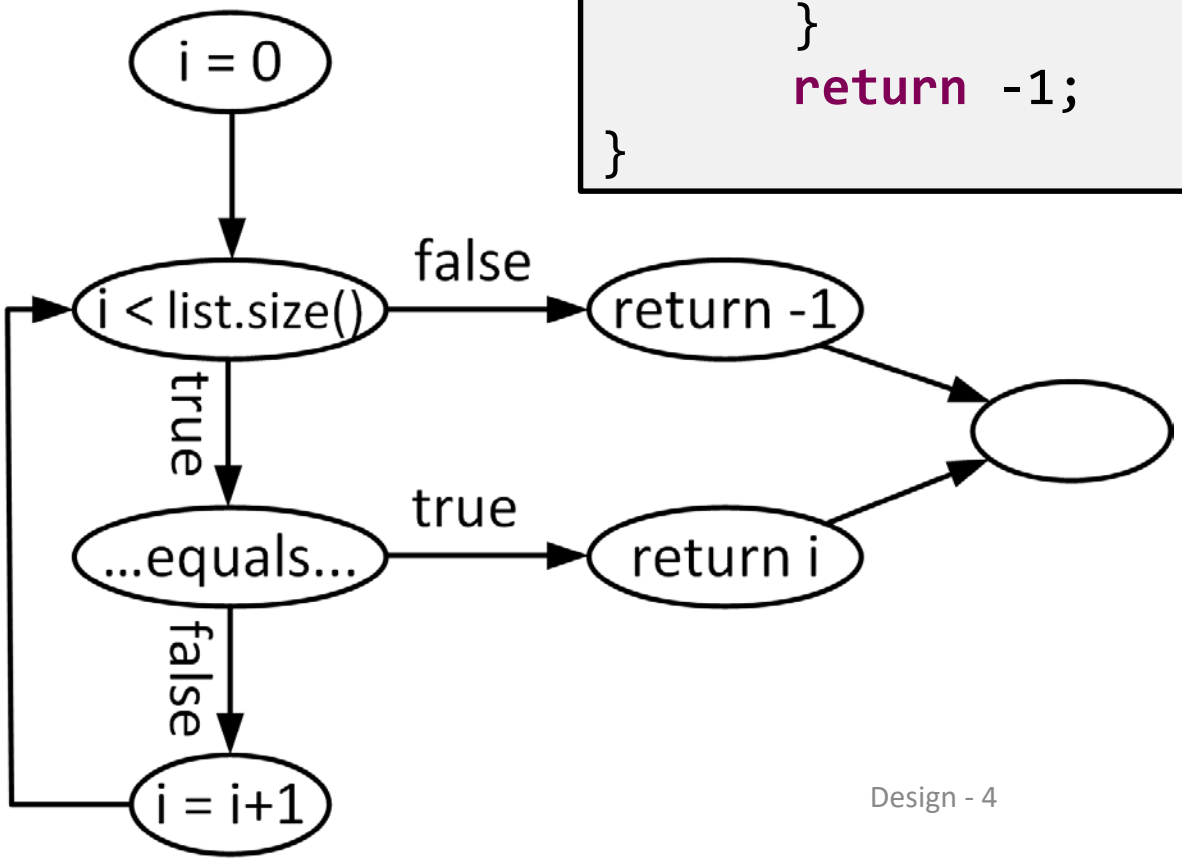


M=n



Example flow graph

```
public int getFirst(List<String> list,
                    String key) {
        int i = 0;
        while (i < list.size()) {
                if (list.get(i).equals(key)) {
                        return i;
                }
                i = i+1;
        }
        return -1;
}
```



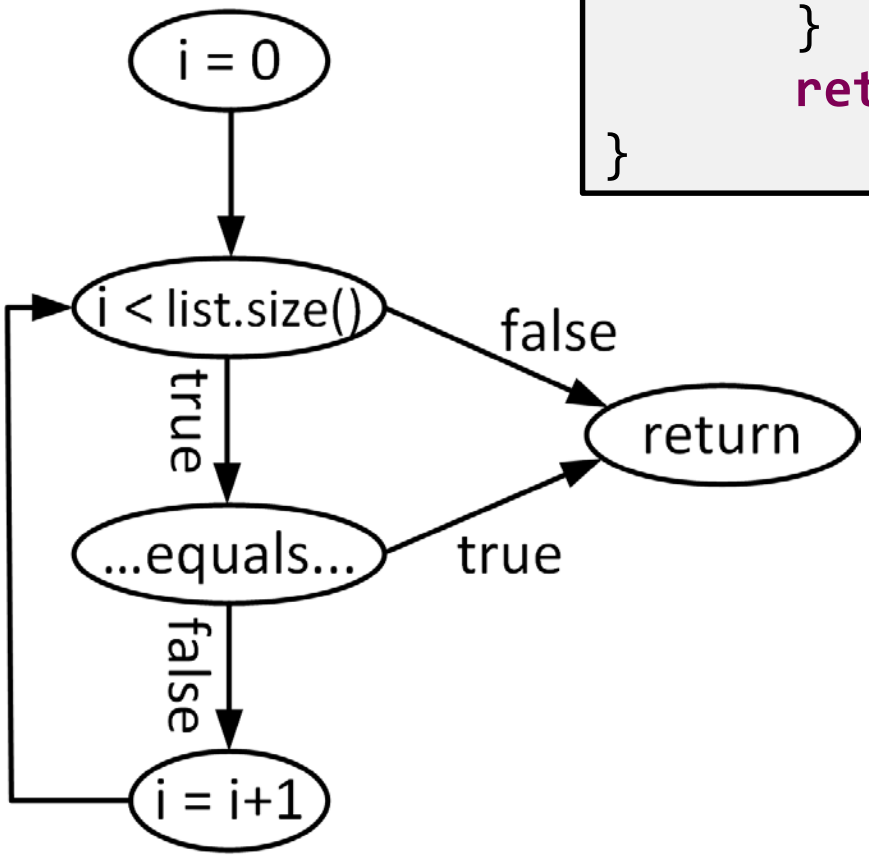
edges = 8

nodes = 7

M = 8 – 7 + 2 = 3

Example flow graph

```
public int getFirst(List<String> list,
                    String key) {
        int i = 0;
        while (i < list.size()) {
                if (list.get(i).equals(key)) {
                        return i;
                }
                i = i+1;
        }
        return -1;
}
```

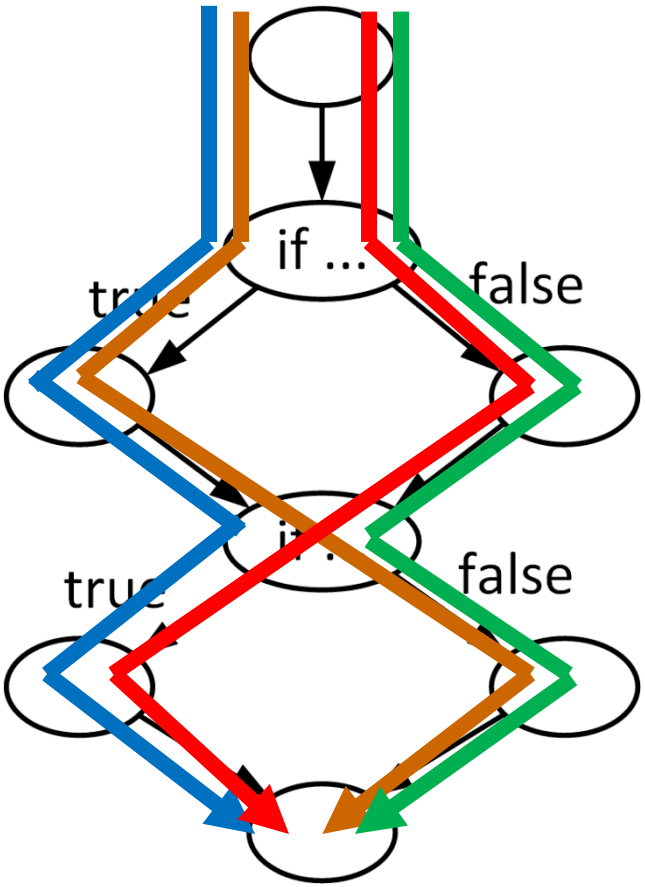


edges = 6

nodes = 5

M = 6 – 5 + 2 = 3

Possible execution paths



#edges = 9

#nodes = 8

M = 9 – 8 + 2 = 3

4 possible execution paths

→ Requires 4 different test cases

Cyclomatic complexity

Definition:

- CC = # edges – # nodes + 2

Theorem:

- In a connected graph with one start node, one end node, and only binary decisions,
  (i.e. decision nodes have 2 outgoing branches)
  # decision nodes = # edges – # nodes + 1

Corollary (assuming binary decisions):

- CC = # branch nodes + 1

Why to keep cyclomatic complexity low?

- Testability:
  desired # test cases = # execution paths ≥ M
- Maintainability:
  - Easier to understand, thus easier to modify

Guideline

- "CC should be lower than 10" [McCabe, 1976]
- "CC should be 5 or lower" [Visser, 2016]

Weighted Methods per Class (WMC)

Overall complexity of a class:

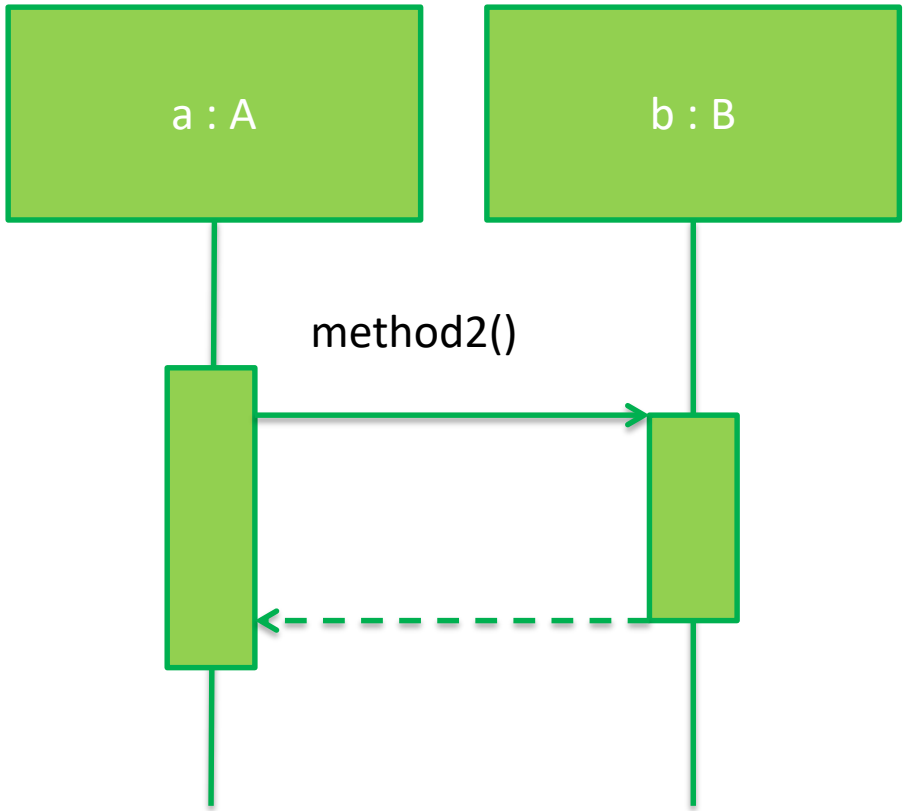
- Sum of cyclomatic complexities of all methods in a class
  - E.g.: if a class has CC values 1, 4, 2, 2, 1 for its (five) methods, then WMC = 10

  i.e., WMC is the number of methods in a class weighted by their complexity
  - If you split a method with CC=6 into two methods, each with CC=3, then WMC does not change…

CA/CE: Coupling between classes

```
class A {
  private B b;
  public void
method1() {
    b.method2();
  }
}
```

```
class B {
  public void
method2() {
    //…
  }
}
```



Coupling between classes

- Class A is coupled to class B when

  Class A is dependent on class B when
  - Methods in A invoke methods declared in B
  - Methods in A access attributes of B
- For B this is called afferent (incoming) coupling
- For A this is called efferent (outgoing) coupling

From Latin: ad = towards ferre = to carry
ex = out of

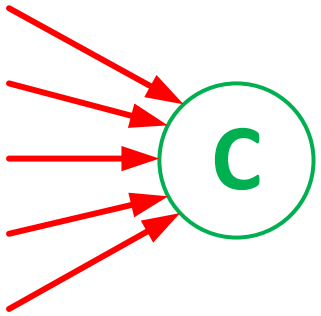
CA: Afferent (incoming) coupling

Afferent coupling CA of a class C =

the number of classes that call methods (and/or access attributes) of C

or: the number of classes that depend on C

What would be the problem with high CA?



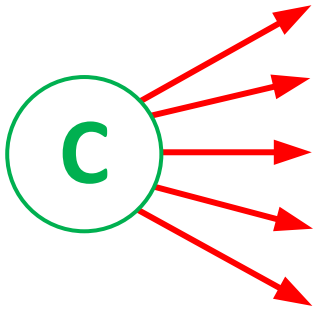
CE: Efferent (outgoing) coupling

Efferent coupling CA of a class C =

the number of classes with methods that are called by C (or have attributes accessed by C)

or: the number of classes on which C is dependent

What would be the problem with high CE?



Instability of a class

- $I = \frac{CE}{CA + CE}$

- High CA, low CE: low I → stable class
- High CE, low CA: high I → instable class

Coupling, related concepts

Coupling between packages (rather than classes):

- CA = # classes outside this package that depend on classes within this package
- CE = # classes inside this package that depend on classes outside this package

Fan-in of a class [Visser 2016] :

- Fan-in of a method: #classes calling this method
- Fan-in of a class: sum of the fan-in of its methods

LCOM: Lack of Cohesion in methods

Informally:

High cohesion in a class:

- Different methods access the same class variables

Low cohesion in a class:

- Different class variables each have their own set of methods that access them

Cohesion: similar methods

- Methods are *similar* when they access common fields
- A cohesive class mainly contains similar methods

```
class C {
  int a;
  int b;
  int c;

  void m1() { a = 1; }
  void m2() { a = 2; b = 3;}
  void m3() { c = 4; }
}
```

Method pair	Intersection of used fields
m1, m2	{a} ***similar***
m1, m3	{} ***dissimilar***
m2, m3	{} ***dissimilar***

LCOM1 [Chidamber & Kemerer, 1994]

LCOM = surplus of dissimilar pairs

- P = number of dissimilar methods pairs
- Q = number of similar method pairs
- If (P > Q) LCOM = P – Q else LCOM = 0

LCOM1, example

```
class C {
  int a;
  int b;
  int c;

  void m1() { a = 1; }
  void m2() { a = 2; b = 3;}
  void m3() { c = 4; }
}
```

Method pair	Intersection of used fields
m1, m2	{a} ***similar***
m1, m3	{} ***dissimilar***
m2, m3	{} ***dissimilar***

- P = **2** (# dissimilar pairs), Q = **1** (# similar pairs)
- LCOM1 = P – Q = **1**

LCOM1, example

```
class C {
  int a;
  int b;
  int c;

  void m1() {a=1;}
  void m2() {a=2; b=3;}
  void m3() {c=4;}
}
```

split

```
class C1 {
  int a;
  int b;

  void m1() {a=1;}
  void m2() {a=2; b=3;}
}
```

P = 0
Q = 1
LCOM = 0

```
class C2 {
  int c;

  void m3() {c=4;}
}
```

P = 0
Q = 0
LCOM = 0

Not always possible or desirable…

LCOM2 (used by Eclipse) [Henderson et al, 1996]

Let a class have

- a attributes $A_1 \ldots A_a$
- m methods $M_1 \ldots M_m$ which access attributes
  (methods that do not access any attribute don't count)
- mA_k = # methods that access A_k
- avg(mA) = average mA_k for k=1..a $= \frac{1}{a} \sum_{k=1..a} mA_k$
- LCOM2 = $\frac{m - \text{avg}(mA)}{m - 1}$

Maximally coherent class:

```
class A {
  int a;
  int b;
  int c;

  void m1() {a = b+c;}
  void m2() {b = a-c;}
  void m3() {c = b-a;}
}
```

$a = 3$, $m = 3$, avg(mA) = 3

LCOM = 0

(m – avg(mA)) / (m – 1) =
(3 – 3) / (3 – 1) = 0

Maximally incoherent class:

```
class B {
  int a;
  int b;
  int c;

  void m1() {a = 1;}
  void m2() {b = 2;}
  void m3() {c = 3;}
}
```

$a = 3$, $m = 3$, avg(mA) = 1

LCOM = 1

(m – avg(mA)) / (m – 1) =
(3 – 1) / (3 – 1) = 1

Other metrics

(E.g.)

DIT Depth of Inheritance Tree

[Chidamber & Kemerer, 1994]

PAR Number of parameters of a method

[Visser 2016]

Related concepts

- Legacy

  Antiquated systems that are hard / impossible to maintain and hard to replace

Related concepts

- **Refactoring**

  Improving the structure of a system / package / class without changing its functionality

- **Technical Debt**

  Postponing refactoring (of which you know it ought to happen) creates “technical debt”.

  Like with financial debts you do pay interest…

Code Smells

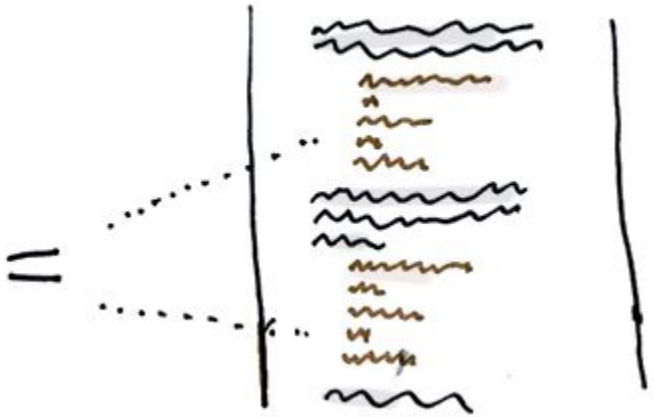
Code patterns that violate best practices,
potential sources of bad maintainability

List at Wikipedia:

- Action at a distance
- Blind faith
- Boat anchor
- Cargo cult programming
- Duplicated code
- Feature Envy
- God class
- Inappropriate intimacy
- Lava flow
- Magic numbers
- Shotgun surgery
- Spaghetti/Lasagna code

Code Smell: Duplicated code

- Two code fragments look almost identical
  - If the same code is found in two or more methods in the same class:
    - Create a new method (in Eclipse: EXTRACT METHOD)
    - call the method from both places

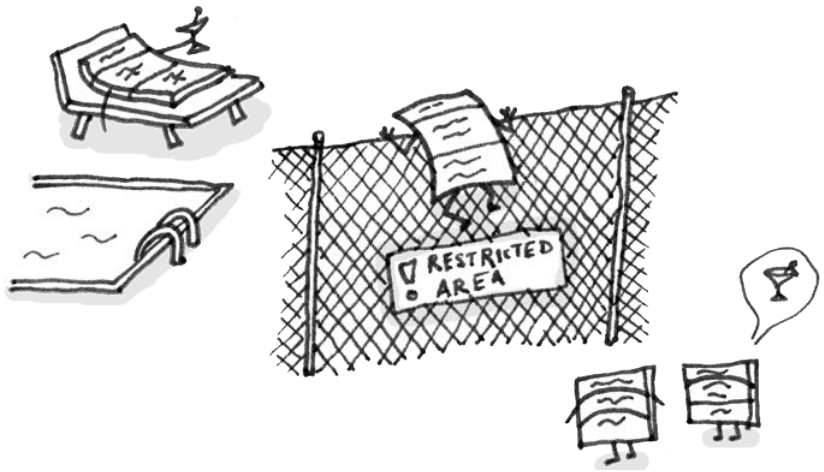


Source: https://sourcemaking.com/refactoring/smells

Code Smell: Feature Envy

- A class excessively uses methods of another class

```
public class Customer { …
   private Phone mobilePhone;
   public String getMobilePhoneNumber() {
      return "(" +
         mobilePhone.getAreaCode() + ") " +
         mobilePhone.getPrefix() + "-" +
         mobilePhone.getNumber();
   }
}
```

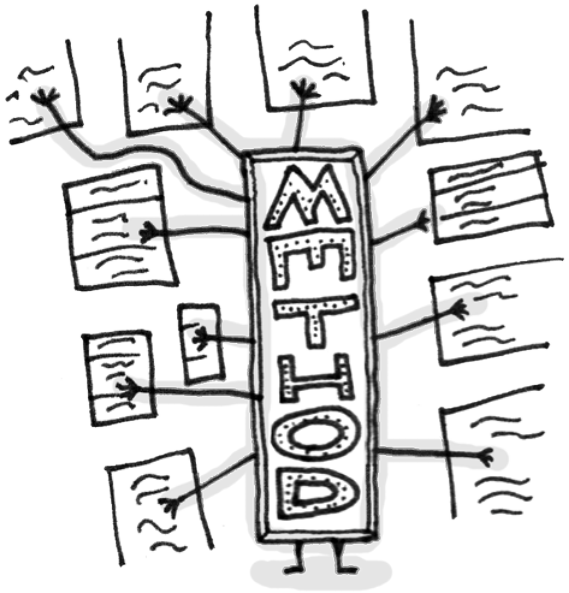


Source: https://sourcemaking.com/refactoring/smells

  - How to repair?
  - Move the method to Phone

Code Smell: Inappropriate intimacy

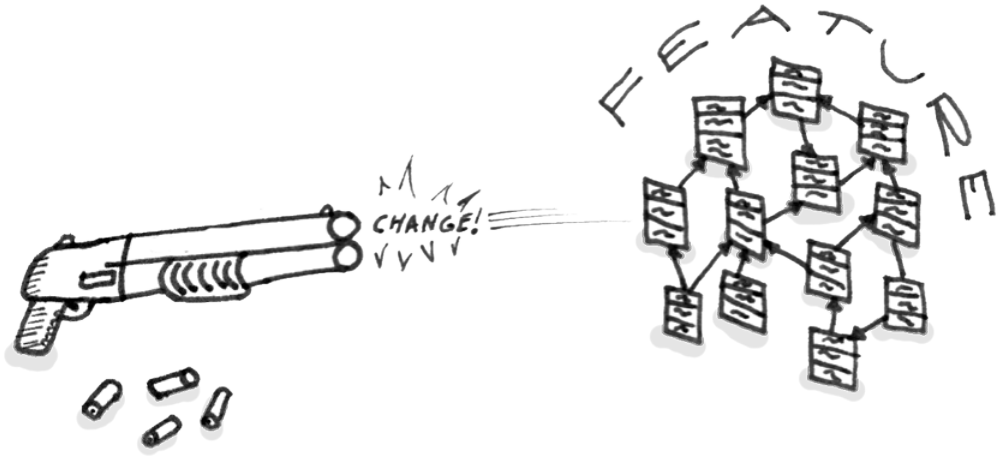
- One class uses attributes and methods of another class that should be private
  - Good classes should know as little about each other as possible
  - Such classes are easier to maintain and reuse



Source: https://sourcemaking.com/refactoring/smells

Code Smell: Shotgun surgery

- Modifications require that you make many small changes to many different classes
  - A single responsibility has been split up among a large number of classes



Source: https://sourcemaking.com/refactoring/smells

Further Reading

- J. Visser (2016): *Building Maintainable Software – Ten Guidelines for Future-Proof Code*. O'Reilly, Sebastopol, CA., USA
  - Based on field work of *Software Improvement Group (SIG)*
  - PDF available for UT students by courtesy of SIG

    (Canvas > Design > Week 4)

  Not required for the test, but **highly recommended** for your personal development if you want to become a professional software developer

Summary

- In the real world, most effort in software development is in fact maintenance
- Building maintainable software pays off
  - Bugs are fewer and easier to find
  - The software is easier to adapt and to extend
    - *For others, as well as for yourself*
- Software metrics can give an indication of suboptimal code
  - *but should be used with care*

Lab Session

- D-4.1 Working with the Eclipse plugin, various introductory exercises
- D-4.2 Lack of Cohesion in Methods
- D-4.3 Cyclomatic Complexity

About signing off

- Last lab session is tomorrow, Wed 5 Dec
- You still can still sign off everything during the project sessions Fri 7 Dec and Mon 10 Dec.
- *If you still need to sign off a lot of design exercises,* ***please do so this week****.*
  On Monday we may not have the capacity to help persons who need to sign off multiple lab sessions. Priority will be given to week 4, then week 3, etc., in backwards order.
- Monday evening the participants list for the test will be made. You need to have finished to be on that list.

Design project

- First deadline: Sun 16 dec 23:59
  - 0.5 point bonus if you meet the deadline
  - The reason to encourage you to meet this deadline: In January you won't have time, you have a programming project

- If you have lost a member of the Design group, contact me for a slightly reduced task description