Metrics Definitions

Scope

Lines of Code

The total number of lines of executable code (i.e., not including comments) in the software program or module being measured

Cyclomatic Complexity

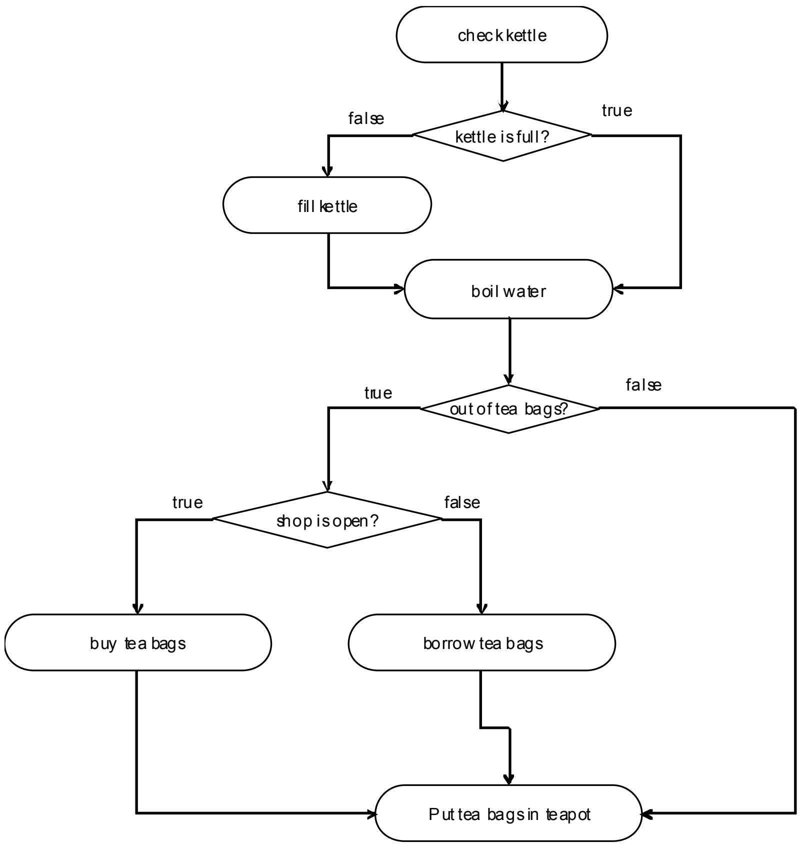
<u>Definition #1:</u> The number of executable paths in a process – for example, distinct paths in a segment of code, or through a use case.

<u>Definition #2:</u> **Cyclomatic Complexity V(G) = E – N + p**

Where **E** is the number of edges in a graph, **N** is the number of nodes in the graph and **p** is the number of connected components.

Example:

By inspection, there are 6 distinct paths.



Halstead Volume

Halstead Volume **$V = N * Log(n)$**

Where N = program length = total number of operators N1 + total number of operands N2, and **n = program vocabulary = number of distinct operators n1 + number of distinct operands n2**

Function Points

Scope measures such as lines of code, Cyclomatic Complexity and Halstead Volume can be applied to functional specifications. For example, they can be applied to acceptance test scripts to help estimate the complexity of the features they are designed to test, or to use case scenarios.

Quality

Defects per Thousand Lines of Code or per Function Point

Effective bug tracking can help build a record of the known bugs within a system. When applied against measures of software scope (e.g., lines of code or function points), they can give an indication of the "buggy-ness" of the software.

SEI Maintainability Index

Maintainability **$M = 171 - 5.2 * \log_2(aveV) - 0.23 * aveV(g') - 16.2 * \log_2 (aveLOC) + 50 * \sin(\sqrt{2.4 * perCM})$**

The coefficients are derived from actual usage. The terms are defined as follows:

aveV = average Halstead Volume V per module

aveV(G) = average cyclomatic complexity per module

aveLOC = the average count of lines of code (LOC) per module; and, optionally

perCM = average percent of lines of comments per module

Variations:

#1: Some implementations omit comments (**perCM**)
#2: Some implementations use extended cyclomatic complexity (**V(G')**)

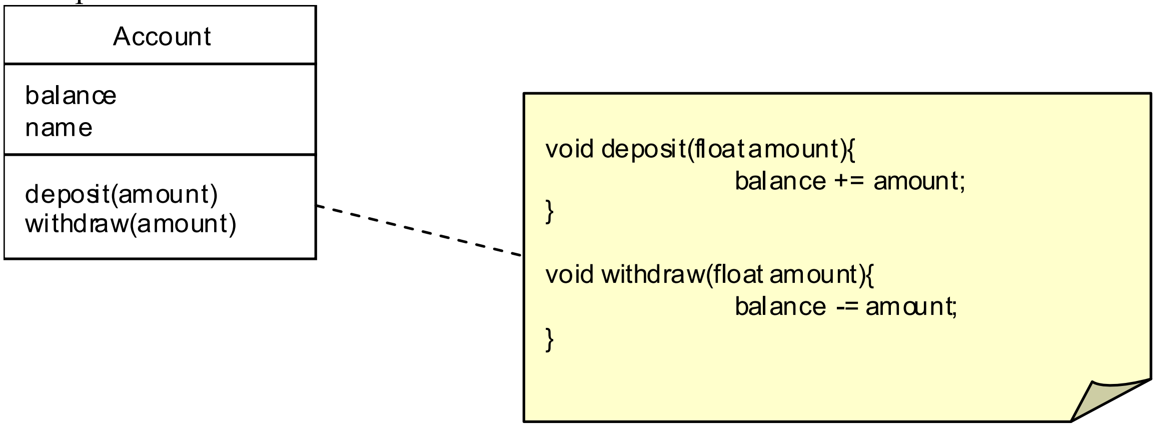
Lack of Cohesion of Methods

LCOM is a measure of how close methods are to the data they access. The more attributes of a class each method accesses (or, more specifically, the more methods that access each attribute), the lower the value of LCOM and the more cohesive the class is said to be. It's calculated by taking the average number of methods accessing the attributes of a class, minus the total number of methods of that class, divided by 1 minus the number of methods.

$LCOM = ((1/a * \sum A) – m)/(1 – m)$

Where **a** is the number of attributes of the class, $\sum$ **A** is the sum across the set of attributes of the number of methods that access each attribute, and **m** is the number of methods of the class.

Example:



```
void deposit(float amount){
                balance += amount;
}

void withdraw(float amount){
                balance -= amount;
}
```

For example, the *Account* class shown above has two attributes (a = 2) which make up the set *{balance, name}*. It has 2 methods (m = 2). The attribute *balance* is accessed by 2 methods (A = 2), and the attribute *name* is accessed by none. So, the Lack of Cohesion of Methods of the *Account* class is:

LCOM = ((½ * (2 + 0)) – 2)/(1 – 2) = -1/-1 = 1

That is to say that the Account class is not cohesive because the attribute name is not accessed at all and therefore arguably doesn't belong in the Account class. If we add methods, getName() and setName() that return and change the value of the name attribute respectively, we get:

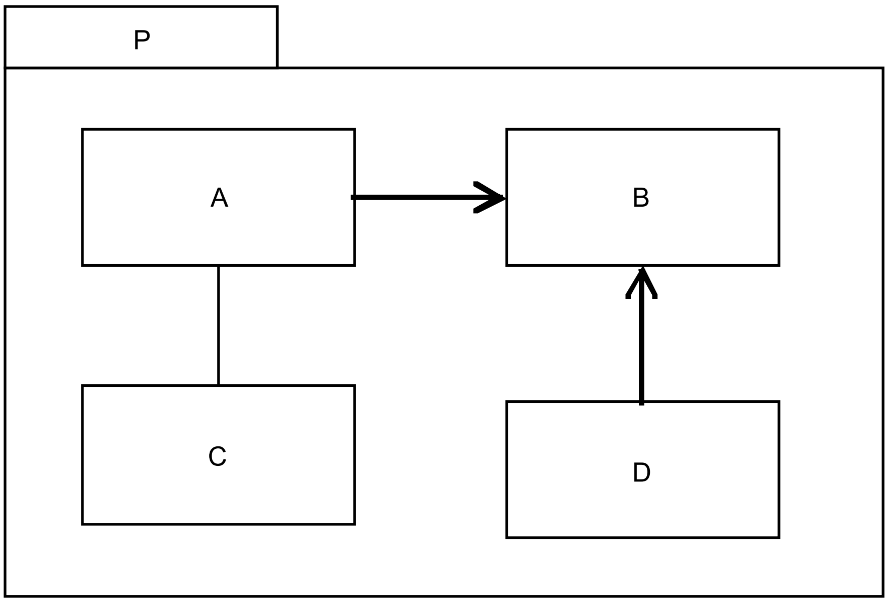
LCOM = ((1/2 * (2 + 2)) – 4)/(1 – 4) = -2/-3 = 0.67

Meaning the Account class is more cohesive. If all the methods accessed both attributes, LCOM would be ((1/2 * (4 + 4)) – 4)/(1-4) = 0. To be *totally* cohesive, a class must therefore have every attribute accessed by every method.

Package Metrics

Cohesion

Highly dependent modules should be packaged together, so they can be released, modified and reused together. Ideally, every module in a package should depend upon every other module in a package, and very little on modules outside the package.



Module A depends directly or indirectly on modules B and C. There are 3 other modules excluding A in package P, so cohesion with respect to module A is 2/3. We can find the modules A depends on by following (navigating) the dependencies. Some dependencies can only be navigated in one direction (denoted by an arrow -> in the direction we can navigate). Other dependencies can be navigated in both directions. These have no arrows at either end.

Cohesion of package P is the average of cohesion with respect to all modules in P, which is:

(2/3 + 0/3 + 1/3 + 2/3) / 4 = 0.42

Instability

- **Afferent couplings (C_a)** – the number of modules in other components that depend on modules in this component
- **Efferent couplings (C_e)** – the number of modules in other components this modules in this component depends on

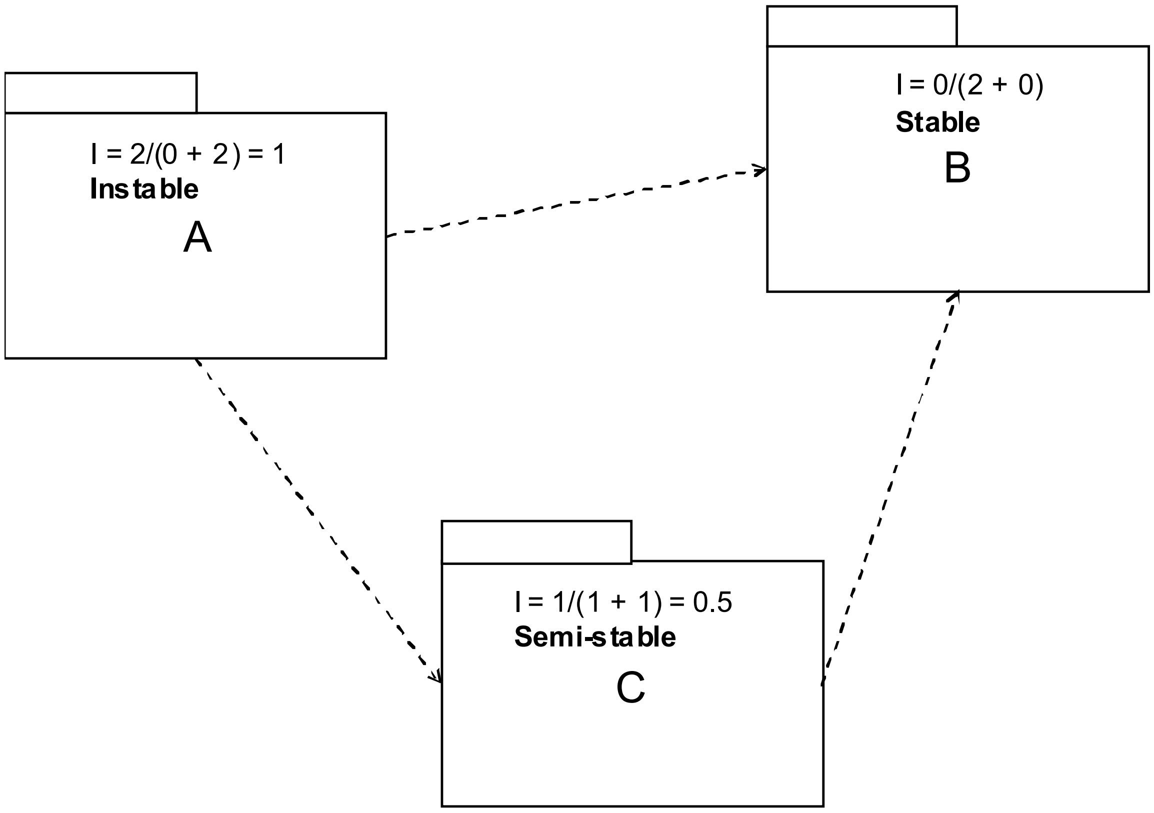
Instability, $\mathbf{I = C_e / (C_a + C_e)}$

The more a component depends on other components, the more possible reasons it might have to need to change. Therefore, a more stable component is one upon which many components depend, but that depends on as few components as possible.

When I = 0, the component is said to be *maximally stable*. This requires zero efferent couplings ($C_e = 0$) but at least one afferent coupling ($C_a > 0$), so components that have no dependencies on other components but upon which other components depend are the most stable.

When I = 1, the component is said to be *maximally instable*. This requires zero afferent couplings ($C_a = 0$) but at least one efferent coupling ($C_e > 0$), so components that have no other components that depend upon them but that depend on other components are the least stable.

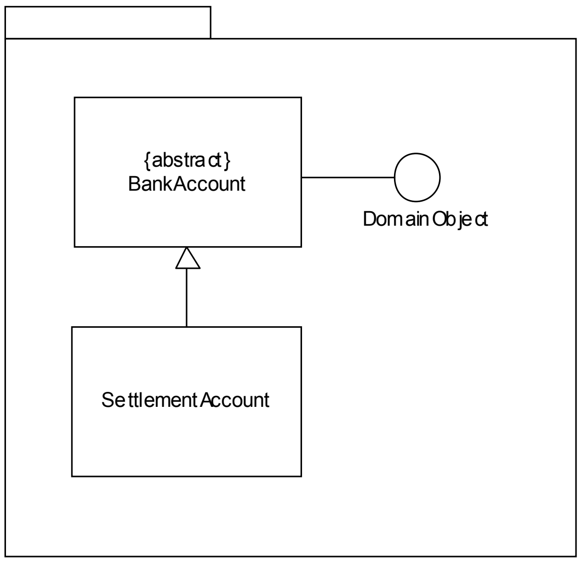
Example:



Abstractness

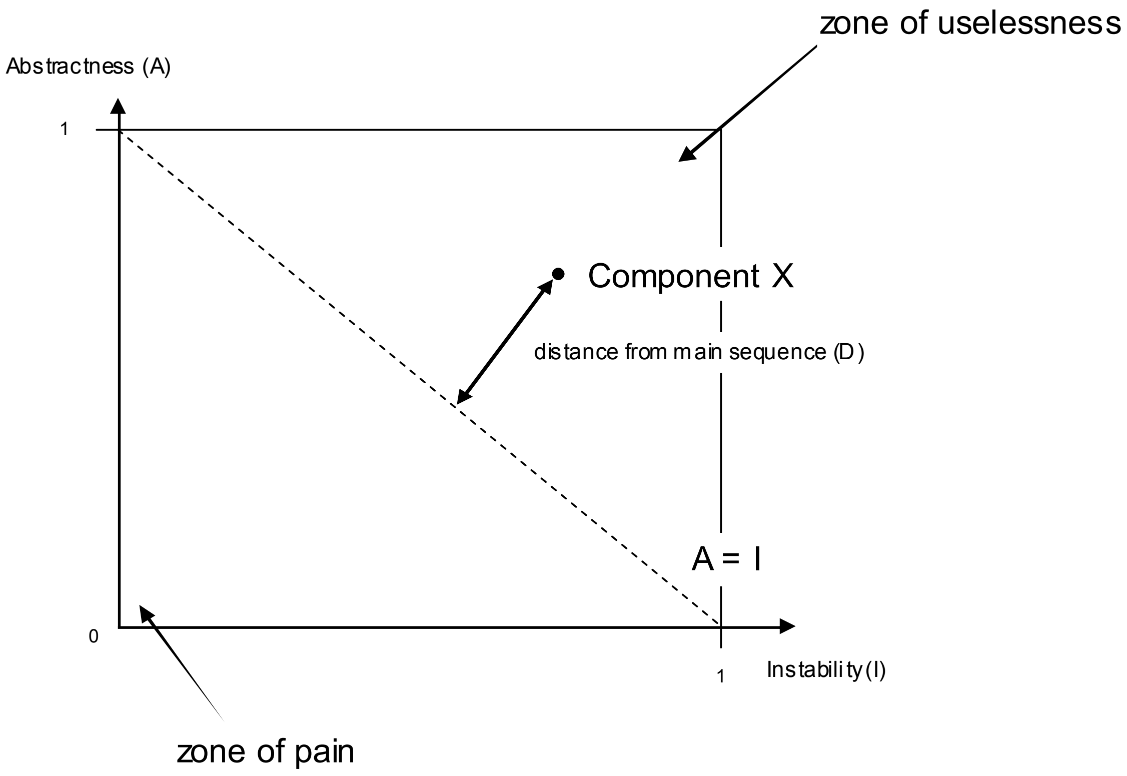
Abstractness **A = number of abstract types/total number of types**

Example:



abstractness = 2/3

Distance from Main Sequence



Distance from main sequence, $\mathbf{D = |(A + I - 1)|/\sqrt{2}}$

Cost

Extrapolated Cost of Task

Extrapolated cost **CE = (development cost of task / development cost of all tasks) * total project cost**

This can be applied per iteration, per release or per project.