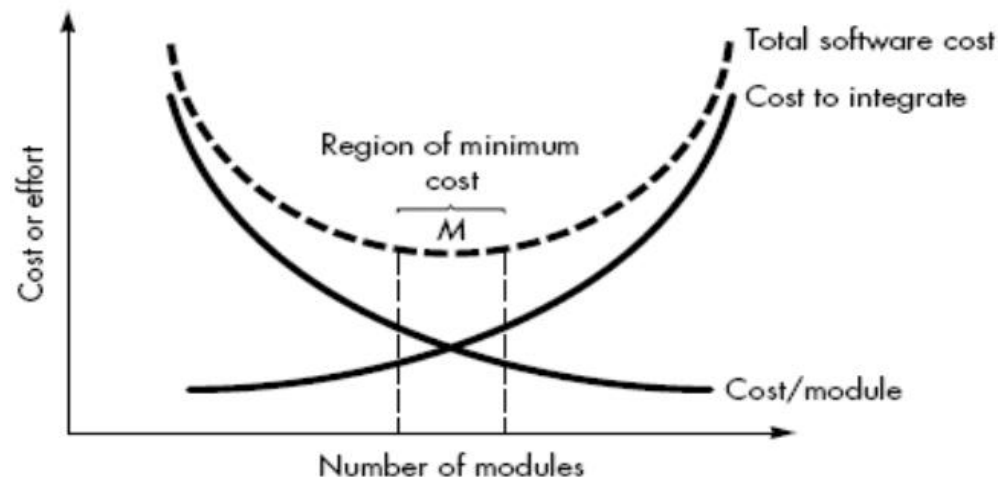


# Software Engineering

Dr. Young-Woo Kwon

# Modularity

- Software is divided into separately named and addressable components, sometimes called modules, that are integrated to satisfy problem requirements

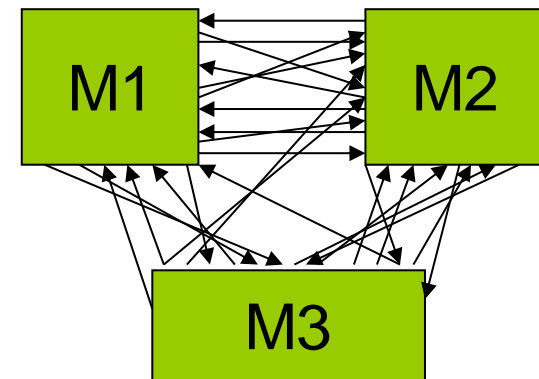


# Cohesion & Coupling

- Cohesion
  - The degree to which the elements of a module belong together
  - A cohesive module performs a single task requiring little interaction with other modules
- Coupling
  - The degree of interdependence between modules
- High cohesion and low coupling

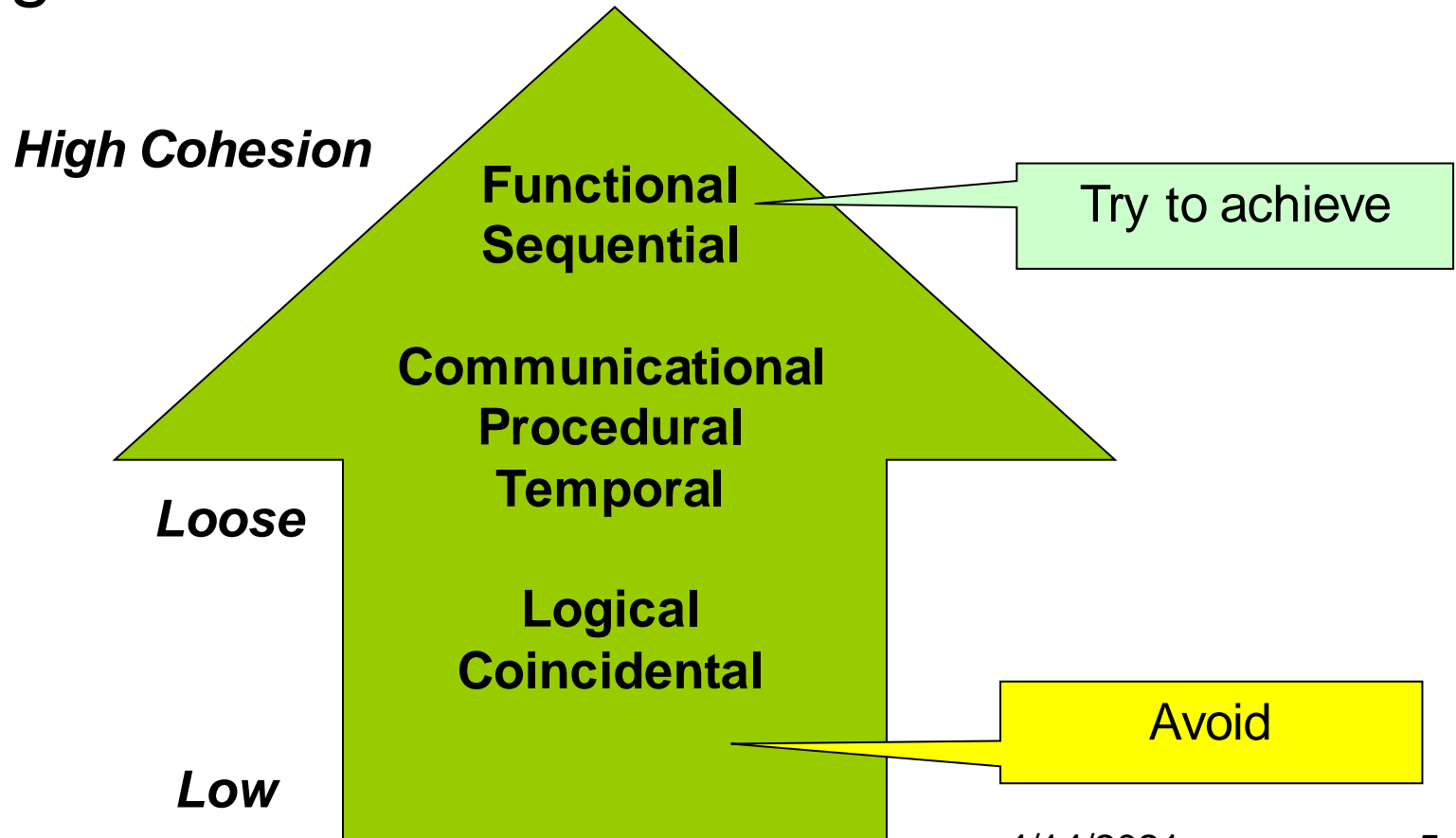
# Characteristics of Good Design

- Component independence
  - Minimize **coupling** between modules
  - Maximize **cohesion** within modules
- Exception identification and handling
- Fault prevention and fault tolerance
- Design for change



# Cohesion

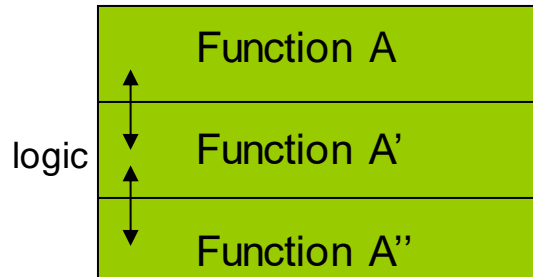
- Degree of interaction **within** module



# Examples of Cohesion

Function A	
Function B	Function C
Function D	Function E

*Coincidental*  
Parts unrelated



*Logical*  
Similar functions

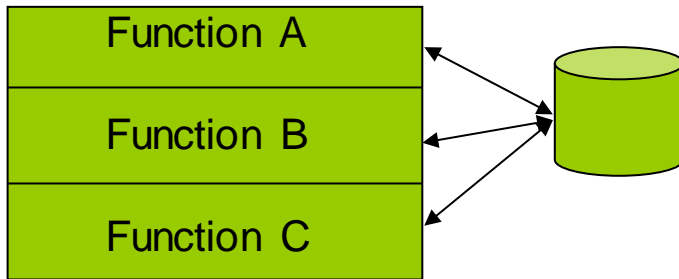
Time $t_0$
Time $t_0 + X$
Time $t_0 + 2X$

*Temporal*  
Related by time

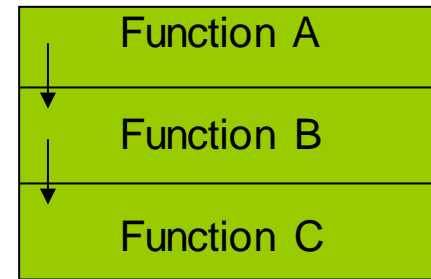
Function A
Function B
Function C

*Procedural*  
Related by order of functions

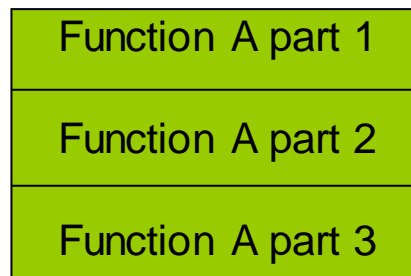
# Examples of Cohesion (Cont.)



*Communicational*  
Access same data

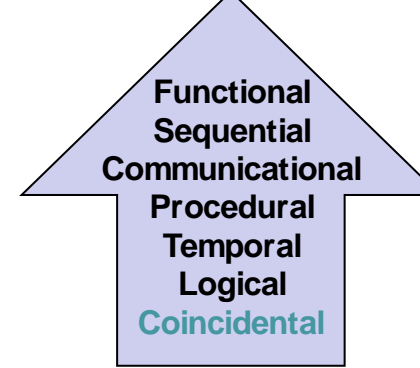


*Sequential*  
Output of one is input to another



*Functional*  
Sequential with complete, related functions

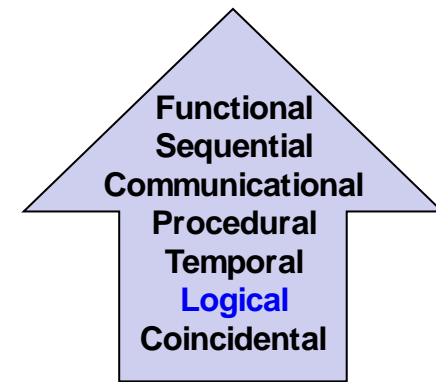
# Coincidental Cohesion



- Def: Parts of the component are unrelated (unrelated functions, processes, or data)
- Parts of the component are only related by their location in source code.
- Elements needed to achieve some functionality are scattered throughout the system.
- Accidental
- Worst form

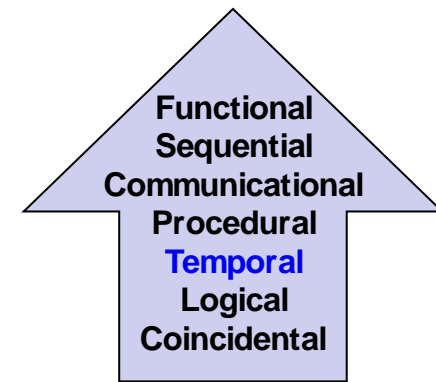


# Logical Cohesion



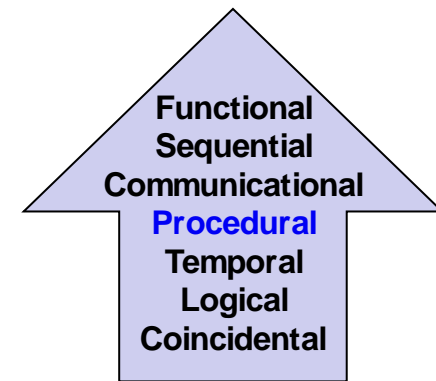
- Def: Elements performs similar activities as selected from outside module
- E.g., Body of function is one huge if-else/switch on operational flag

# Temporal Cohesion



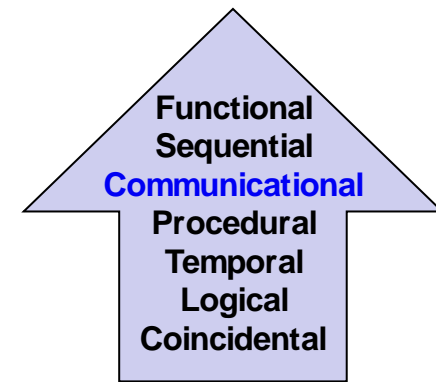
- Def: Elements are related by timing involved
- Elements are grouped by when they are processed.
- Example: An exception handler that
  - Closes all open files
  - Creates an error log
  - Notifies user
  - Lots of different activities occur, all **at same time**

# Procedural Cohesion



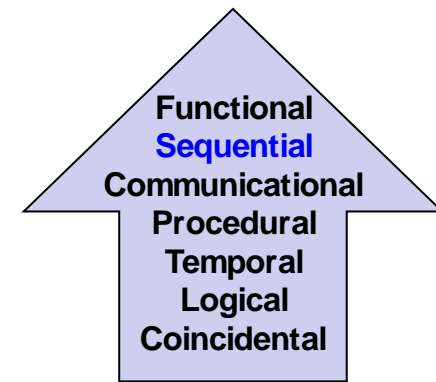
- Def: Elements of a component are related only to ensure a particular order of execution.
- Actions are still weakly connected and unlikely to be reusable.
- Example:
  - ...
  - Write output record
  - Read new input record
  - Pad input with spaces
  - Return new record
  - ...

# Communicational Cohesion



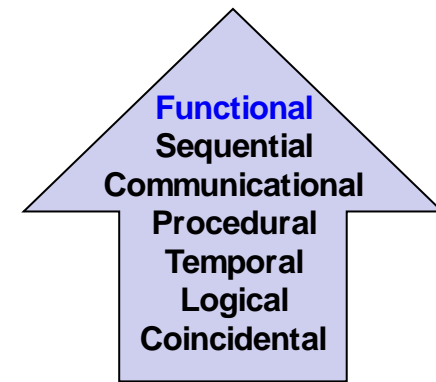
- Def: Unrelated operations on the same data or to produce the same data.
- Examples:
  - Update record in database and send it to the printer
    - Update a record on a database
    - Print the record
  - Fetch unrelated data at the same time.
    - To minimize disk access

# Sequential Cohesion



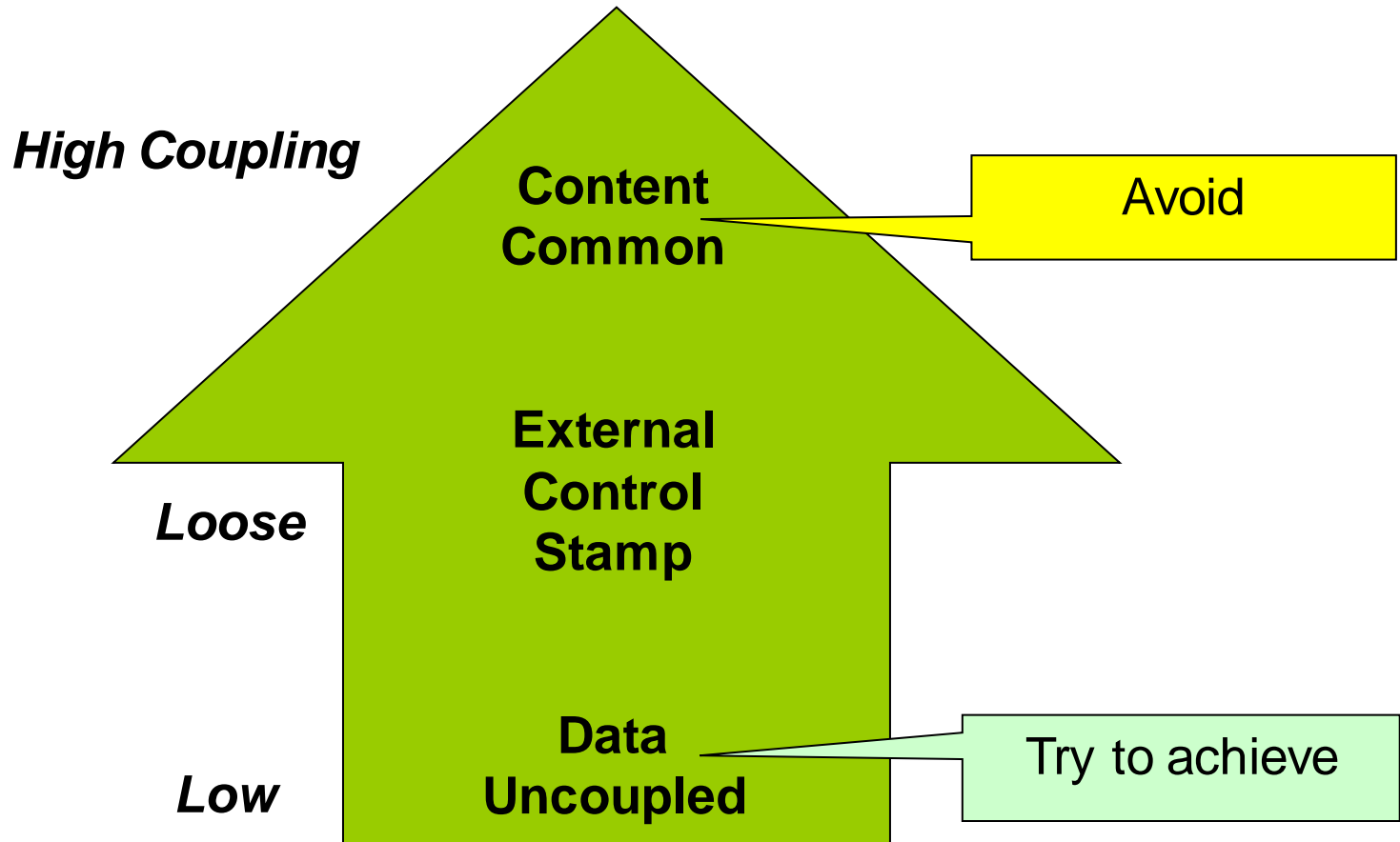
- Def: The output of one part is the input to another.
- *Data flows* between parts (different from procedural cohesion)
- Occurs naturally in functional programming languages

# Functional Cohesion

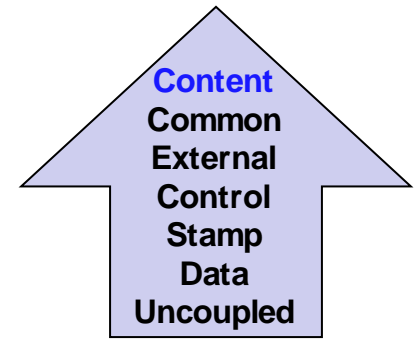


- Def: Every essential element to a single computation is contained in the component.
- Every element in the component is essential to the computation.
- Ideal situation
- What is a functionally cohesive component?
  - One that not only performs the task for which it was designed but
  - it performs only that function and nothing else.

# Type of Coupling



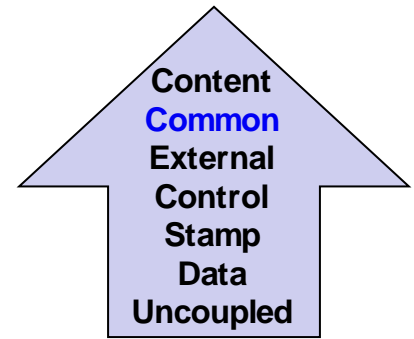
# Content Coupling



- Def: One component modifies another.
- Example:
  - Component directly modifies another's data
  - *Component modifies another's code, e.g., jumps into the middle of a routine → Nearly impossible*

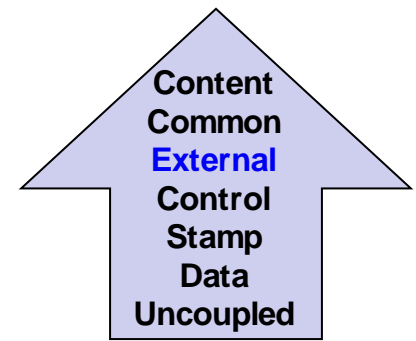


# Common Coupling



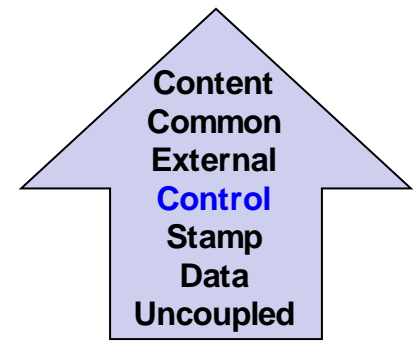
- Def: More than one component share data such as global data structures
- Usually a poor design choice because
  - Lack of clear responsibility for the data
  - Reduces readability
  - Difficult to determine all the components that affect a data element (reduces maintainability)
  - Difficult to reuse components
  - Reduces ability to control data accesses

# External Coupling



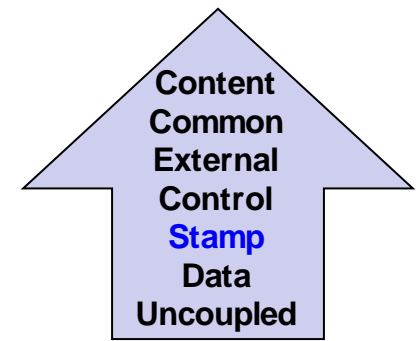
- Def: Two components share something externally imposed, e.g.,
  - External file
  - Device interface
  - Protocol
  - Data format

# Control Coupling



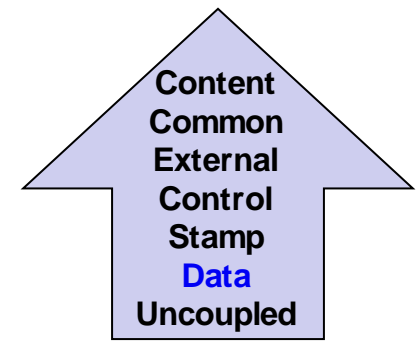
- Def: Component passes control parameters to coupled components.
- Good example: sort that takes a comparison function as an argument.
  - The sort function is clearly defined: return a list in sorted order, where sorted is determined by a parameter.

# Stamp Coupling



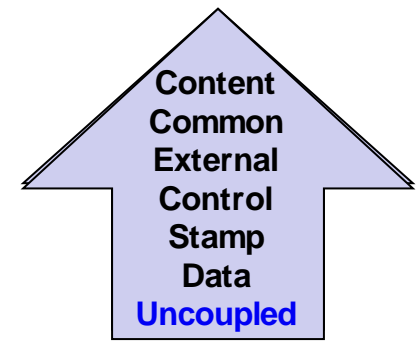
- Def: Component passes a data structure to another component that does not have access to the entire structure.
- Requires second component to know how to manipulate the data structure (e.g., needs to know about implementation).
- The second has access to more information that it needs.
- May be necessary due to efficiency factors: this is a choice made by insightful designer, not lazy programmer.

# Data Coupling



- Def: Component passes data (not data structures) to another component.
- Every argument is simple argument or data structure in which all elements are used
- Good, if it can be achieved.
- Example: Customer billing system
  - The print routine takes the customer name, address, and billing information as arguments.

# Uncoupled



- Completely uncoupled components are not systems.
- Systems are made of interacting components.

# Cohesion

- Degree of interaction **within** module
- Seven levels of cohesion
  - 7. Functional | Informational (best)
  - 6. Sequential
  - 5. Communicational
  - 4. Procedural
  - 3. Temporal
  - 2. Logical
  - 1. Coincidental (worst)

# Information Hiding

- Do not expose internal information of a module unless necessary
  - E.g., private fields, getter/setter methods



# Abstraction

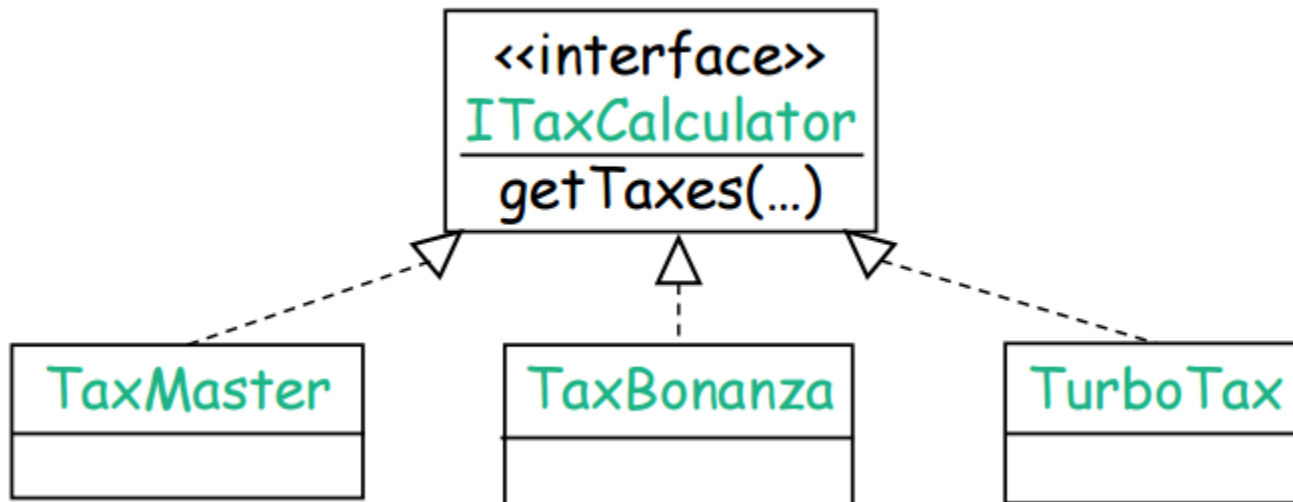
- To manage the complexity of software
- To anticipate detail variations and future changes

# Abstraction to Reduce Complexity

- We abstract complexity at different levels
  - At the highest level, a solution is stated in broad terms, such as “process sale”
  - At any lower level, a more detailed description of the solution is provided, such as internal algorithm of the function and data structure

# Abstraction to Anticipate Changes

- Define interfaces to leave implementation details undecided
- Polymorphism



# How to measure complexity

- Measure certain aspects of the software (lines of code, # of if-statements, depth of nesting, ...)
- Use these numbers as a criterion to assess a design, or to guide the design
- Interpretation: higher value  $\Rightarrow$  higher complexity  $\Rightarrow$  more effort required (= worse design)
- Two kinds:
  - **intra-modular**: inside one module
  - **inter-modular**: between modules

# How to measure complexity

- Measure certain aspects of the software (lines of code, # of if-statements, depth of nesting, ...)
- Use these numbers as a criterion to assess a design, or to guide the design
- Interpretation: higher value  $\Rightarrow$  higher complexity  $\Rightarrow$  more effort required (= worse design)
- Two kinds:
  - **intra-modular**: inside one module
  - **inter-modular**: between modules

# 코드 메트릭 값

2018. 11. 02. • 읽는 데 3분 •    

최신 소프트웨어 응용 프로그램의 복잡성이 높아 안정적이고 유지 관리 가능한 코드 만들기가 늘어납니다. 코드 메트릭은 개발자가 개발 중인 코드에 대해 더 정확히 파악할 수 있도록 하는 소프트웨어 측정 방법입니다. 코드 메트릭을 활용 하여 개발자는 형식 및/또는 메서드는 수정 하거나 더 철저 하 게 테스트를 파악할 수 있습니다. 개발 팀은 잠재적 위험을 식별 하고 프로젝트의 현재 상태를 이해 하고 소프트웨어 개발 하는 동안 진행률을 추적할 수 있습니다.

개발자는 복잡 하고 관리 코드의 유지 관리 편의성을 측정 하는 코드 메트릭 데이터를 생성 하려면 Visual Studio를 사용할 수 있습니다. 전체 솔루션 또는 단일 프로젝트에 대해 코드 메트릭 데이터를 생성할 수 있습니다.

Visual Studio에서 코드 메트릭 데이터를 생성 하는 방법에 대 한 정보를 참조 하세요. [방법: 코드 메트릭 데이터 생성](#)합니다.

## 소프트웨어 측정

다음은 코드를 Visual Studio를 계산 하는 메트릭 결과 보여 줍니다.

- **유지 관리 인덱스** -코드를 유지 하는 상대적인 편의성을 나타내는 0과 100 사이의 인덱스 값을 계산 합니다. 값이 높으면 더 나은 유지 관리를 의미 합니다. 코드에서 문제점을 신속 하 게 식별 하 색으로 구분 된 등급을 사용할 수 있습니다. 녹색 등급 20과 100 사이의 이며 코드에 적절 한 유지 관리에 있음을 나타냅니다. 노란색 등급 10에서 19 사이의 약간 유지 관리 가능한 코드를 나타냅니다. 빨간색 등급을 0에서 9 사이의 등급을 낮은 유지 관리를 나타냅니다. 자세한 내용은 참조는 [유지 관리 인덱스 범위 및 의미](#) 블로그 게시물.
- **순환 복잡성** -코드의 구조적 복잡성을 측정 합니다. 프로그램 흐름에 다른 코드 경로 수를 계산 하 여 생성 됩니다. 복잡 한 제어 흐름에 있는 프로그램을 적절 한 코드 검사를 위해 더 많은 테스트 필요 하며 줄이려면. 자세한 내용은 참조는 [순환 복잡성에 대 한 Wikipedia 항목](#)합니다.
- **상속 수준** -다른 거슬러 올라갑니다 기본 클래스에서 상속 되는 다른 클래스의 수를 나타냅니다. 상속 수준 클래스는 기본 클래스에서 변경 영향을 줄 수는 상속 된 클래스 중 하나에 결합 하는 것과 비슷합니다. 높을수록이 번호, 깊어집니다 상속 및 주요 될 기본 클래스 수정에 대 한 높을수록 잠재적인 변경 합니다. 상속 수준에 대 한 낮은 값 좋은 이며 높은 값이 잘못 되었습니다.
- **클래스 결합** -매개 변수, 지역 변수, 반환 형식, 메서드 호출, 제네릭 또는 템플릿 인스턴스화, 기본 클래스, 인터페이스 구현, 외부 형식에 정의 된 필드를 통해 고유한 클래스 결합을 측정 하고 특성 장식 합니다. 좋은 소프트웨어 설계는 형식 및 메서드 높은 응집력 및 있어야 결합 부족을 나타냅니다. 높은 결합 다시 사용 하고 다른 형식에는 많은 상호 종속성으로 인해 유지 관리 하기 어려울 정도로 디자인을 나타냅니다. 자세한 내용은 참조는 [클래스 결합](#) 블로그 게시물.
- **줄의 코드로** -코드에서 줄의 대략적인 수를 나타냅니다. 수 IL 코드를 기반으로 하며 되지 않으므로 소스 코드 파일에서 줄의 정확한 수를 너무 많은 작업을 수행 하려고 하는 형식 또는 메서드 및 서로 분리 해야 높은 수를 나타낼 수 있습니다. 또한 형식 또는 메서드가 유지 관리 하기가 수 있습니다 나타낼 수 있습니다.

### ① 참고

합니다 [명령줄 버전](#) 코드 메트릭 도구 계산 코드의 실제 줄 IL 대신 소스 코드를 분석 하므로 합니다.

# intra-modular

- attributes of a single module
- two classes:
  - measures based on size
  - measures based on structure

# Size-Oriented Metrics

- Size of the software produced
- *LOC* - Lines Of Code
- *KLOC* - 1000 Lines Of Code
- *SLOC* – Statement Lines of Code (ignore whitespace)
- Typical Measures:
  - Errors/KLOC, Defects/KLOC, Cost/LOC, Documentation Pages/KLOC



# Sized-based complexity measures

- counting lines of code
  - differences in verbosity
  - differences between programming languages
  - $a := b$  versus **while**  $p \neq \text{nil}$  **do**  $p :=$   
 $p$
- Halstead's "software science", essentially counting operators and operands

# Software science basic entities

- $n_1$ : number of unique operators
- $n_2$ : number of unique operands
- $N_1$ : total number of operators
- $N_2$ : total number of operands

# Example program

```
public static void sort(int x []) {  
    for (int i=0; i < x.length-1; i++) {  
        for (int j=i+1; j < x.length; j++) {  
            if (x[i] > x[j]) {  
                int save=x[i];  
                x[i]=x[j]; x[j]=save  
            }  
        }  
    }  
}
```

operator, 1 occurrence

operator, 2 occurrences

operator

# of occurrences

public	1
sort()	1
int	4
[]	7
{}	4
for {;;}	2
if ()	1
=	5
<	2
...	...
<hr/>	
$n_1 = 17$	$N_1 = 39$

# Example program

```
public static void sort(int x []) {  
    for (int i=0; i < x.length-1; i++) {  
        for (int j=i+1; j < x.length; j++) {  
            if (x[i] > x[j]) {  
                int save=x[i];  
                x[i]=x[j]; x[j]=save;  
            }  
        }  
    }  
}
```

operand, 2 occurrences

operand, 2 occurrences

operand

# of occurrences

x	9
length	2
i	7
j	6
save	2
0	1
1	2
<hr/>	
$n_2 = 7$	$N_2 = 29$

# Other software science formulas

- size of vocabulary:  $n = n_1 + n_2$
- program length:  $N = N_1 + N_2$
- volume:  $V = N \log_2 n$
- level of abstraction:  $L = V^* / V$   
approximation:  $L' = (2/n_1)(n_2/N_2)$
- programming effort:  $E = V/L$
- estimated programming time:  $T' = E/18$
- estimate of  $N$ :  $N' = n_1 \log_2 n_2 : n_2 \log_2 n_2$
- for this example:  $N = 68, L = .015$

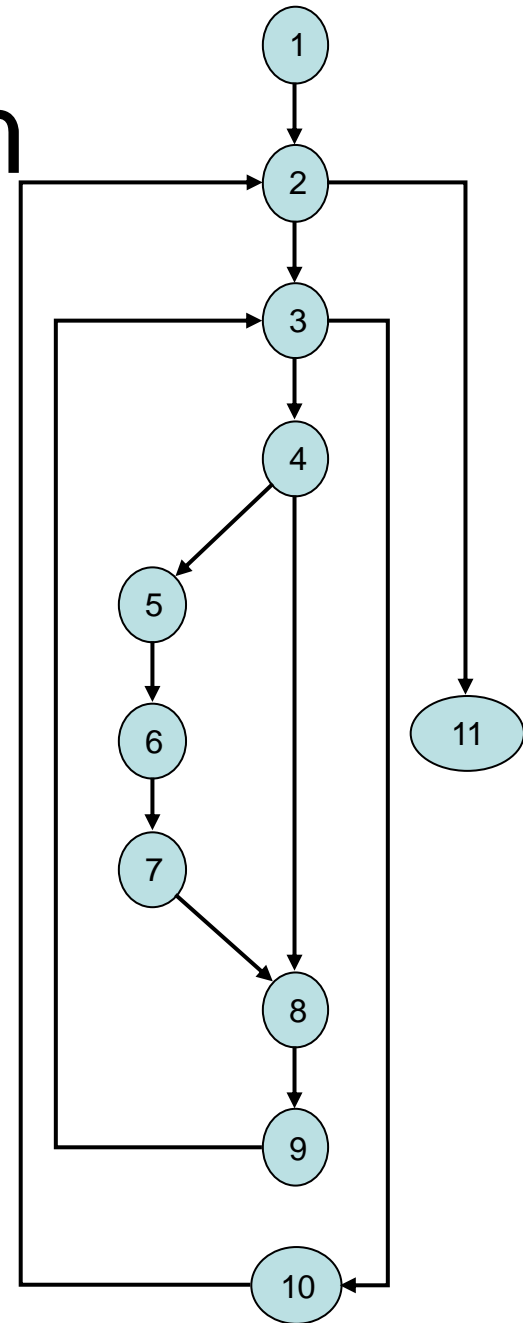
# Structure-based measures

- based on
  - control structures
  - data structures
  - or both
- example complexity measure based on data structures: average number of instructions between successive references to a variable
- best known measure is based on the control structure: McCabe's cyclomatic complexity



# Example program

```
public static void sort(int x []) {  
    for (int i=0; i < x.length-1; i++) {  
        for (int j=i+1; j < x.length; j++) {  
            if (x[i] > x[j]) {  
                int save=x[i];  
                x[i]=x[j]; x[j]=save  
            }  
        }  
    }  
}
```



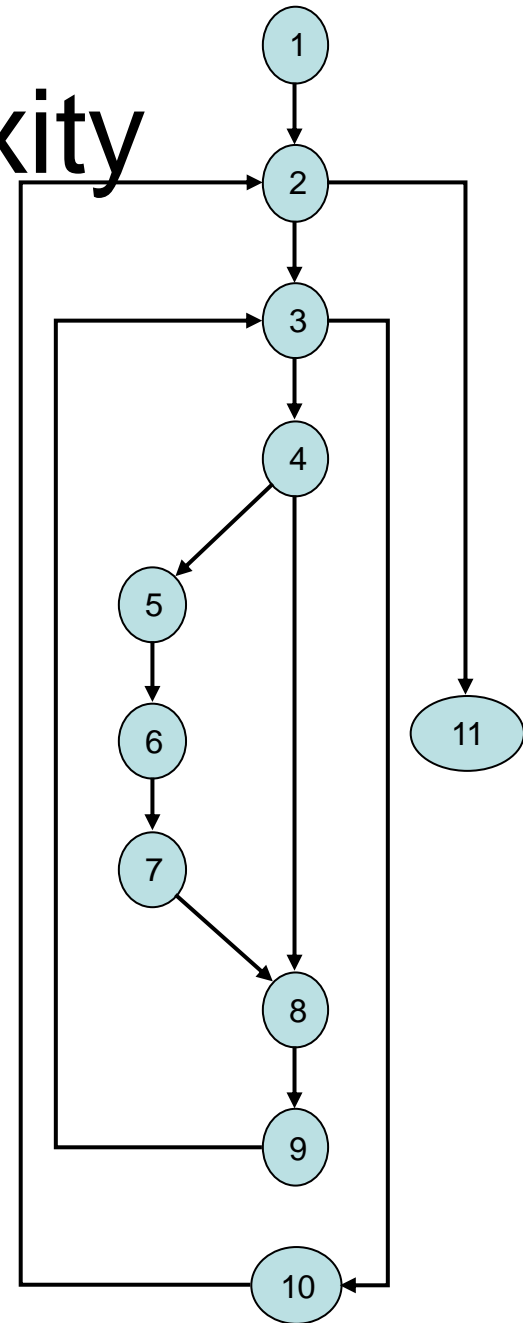
# Cyclomatic complexity

e = number of edges (13)

n = number of nodes (11)

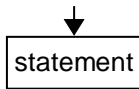
p = number of connected components (1)

$$CV = e - n + p + 1 \quad (4)$$



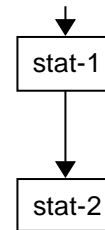
# Issues (1)

Single statement:



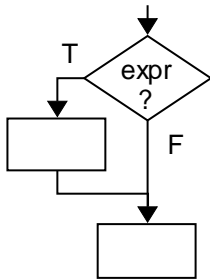
= CC =

Two (or more) statements:



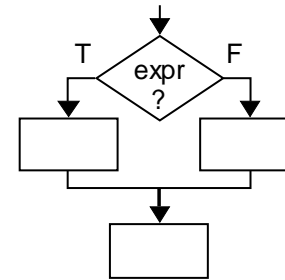
# Issues (2)

Optional action:



= CC =

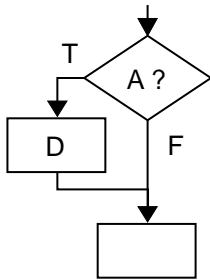
Alternative choices:



# Issues (3)

Simple condition:

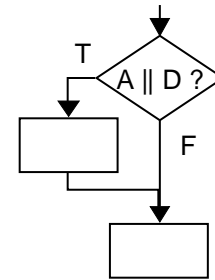
```
if (A) then D;
```



= CC =

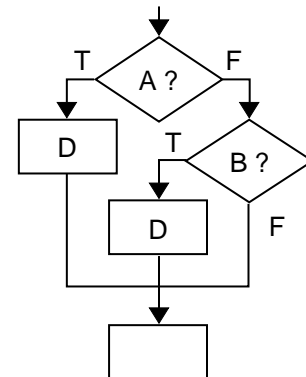
Compound condition:

```
if (A OR B) then D;
```



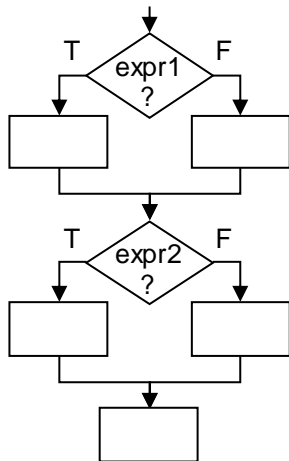
BUT, compound condition can be written as a nested IF:

```
if (A) then D;  
else if (B) then D;
```



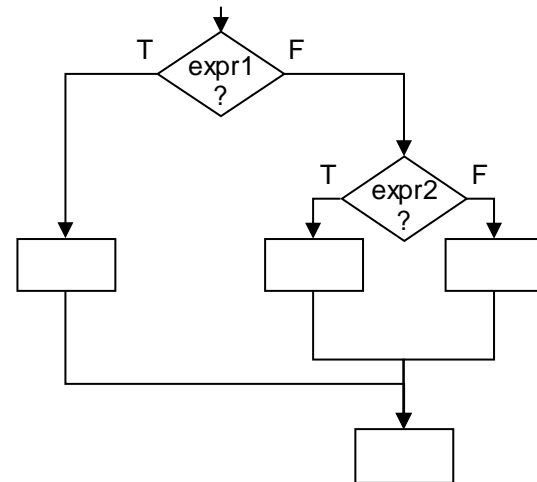
# Issues (5)

Two sequential decisions:



= CC =

Two nested decisions:

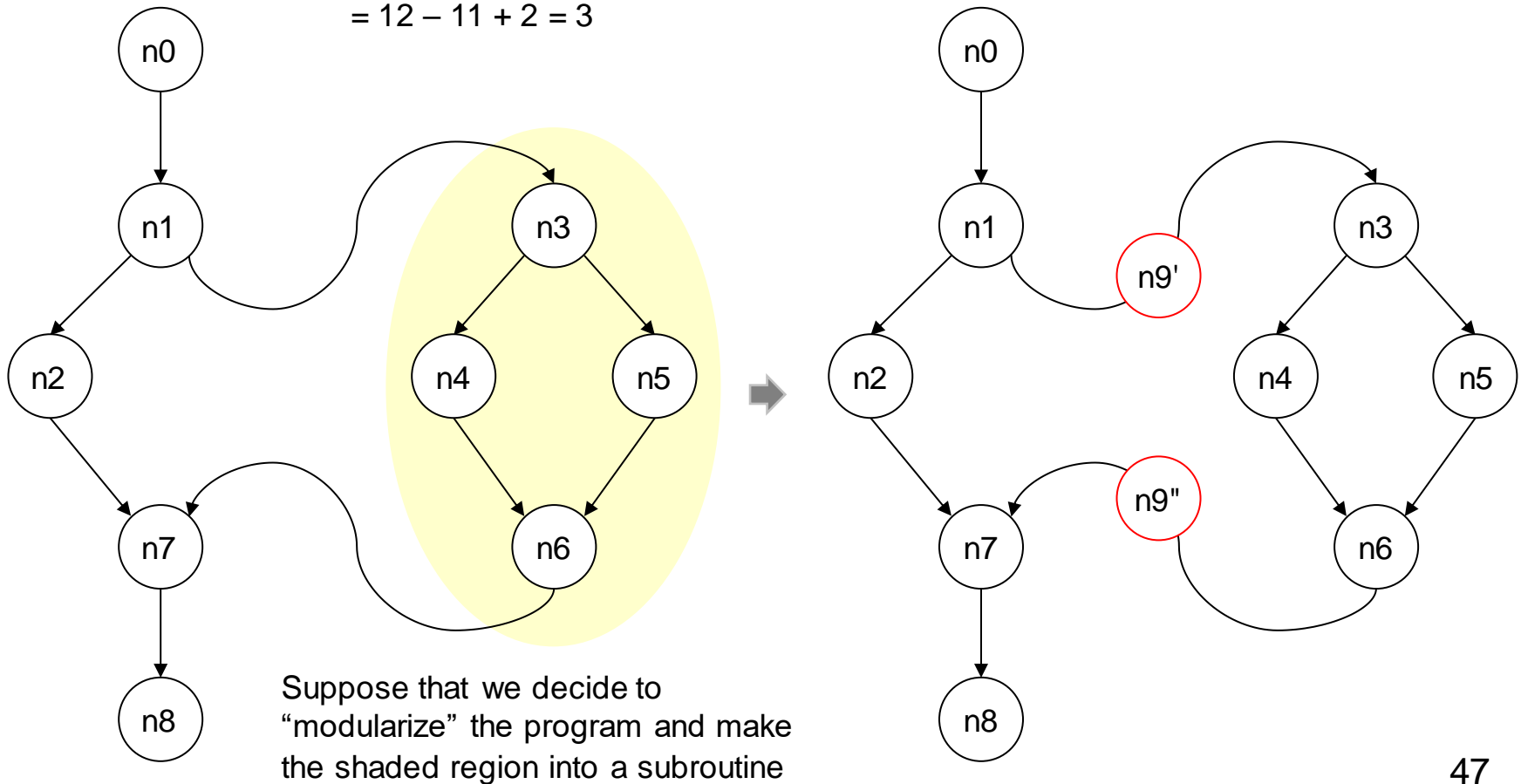


But, it is known that people find nested decisions more difficult ...

# CC for Modular Programs (1)

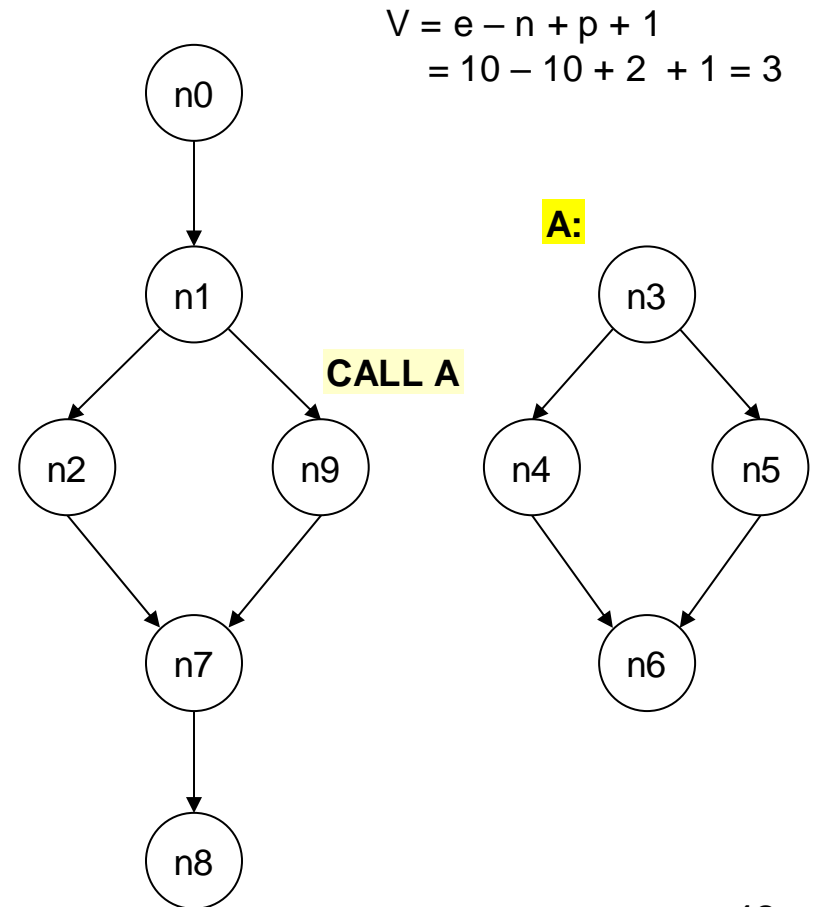
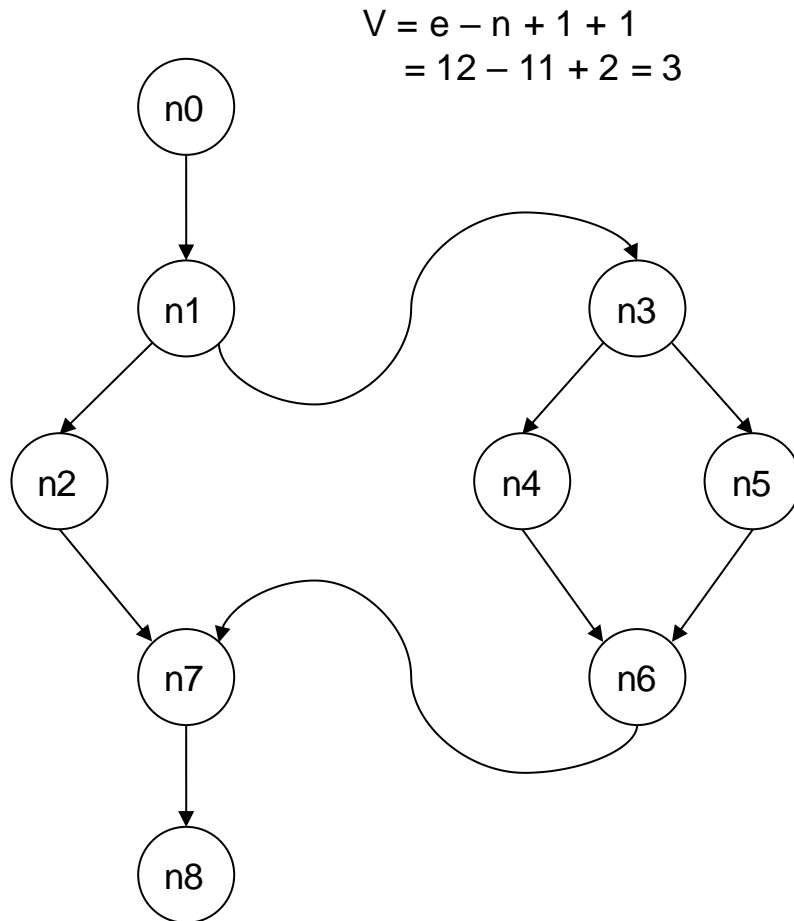
Adding a sequential node  
does not change CC:

$$V = e - n + 1 + 1 \\ = 12 - 11 + 2 = 3$$



# CC for Modular Programs (2)

Modularization should not increase complexity








## Cyclomatic Complexity      Probability of Errors

1~10	5%
20~30	20%
50 이상	40%
거의 100	60%

복잡도 V(G)	위험성 평가
1-10	단순 프로그램, 위험성 낮음
11-20	다소 복잡한 프로그램, 위험성 중간
21-50	복잡한 프로그램, 위험성 높음
51 이상	테스트 불가능한 프로그램, 위험성 매우 높음

# CA1502: 지나치게 복잡하게 만들지 마세요.

2016. 11. 04. • 읽는 데 2분 •    

TypeName	AvoidExcessiveComplexity
CheckId	CA1502
범주	Microsoft.Maintainability
주요 변경 내용	최신이 아님

## 원인

메서드에 과도 한 순환 복잡성이 있습니다.

## 규칙 설명

순환 복잡성은 조건부 분기의 수와 복잡성에 따라 결정 되는 메서드를 통해 선형 독립적 경로의 수를 측정 합니다. 일반적으로 낮은 순환 복잡성은 이해, 테스트 및 유지 관리가 쉬운 방법을 나타냅니다. 순환 복잡성은 메서드의 제어 흐름 그래프에서 계산 되며 다음과 같이 제공 됩니다.

순환 복잡성 = 가장자리 수-노드 수 + 1

노드 는 논리 분기를 나타내며 *가장자리* 는 노드 사이의 선을 나타냅니다.

이 규칙은 순환 복잡성이 25 이상일 때 위반을 보고 합니다.

[관리 코드의 복잡성을 측정](#) 하 여 코드 메트릭에 대해 자세히 알아볼 수 있습니다.

## 위반 문제를 해결하는 방법

이 규칙 위반 문제를 해결 하려면 메서드를 리팩터링하여 순환 복잡성을 줄입니다.

## 경고를 표시 하지 않는 경우

복잡성을 쉽게 줄이고 메서드를 이해, 테스트 및 유지 관리할 수 있는 경우에는이 규칙에서 경고를 표시 하지 않는 것이 안전 합니다. 특히, large `switch` (`Select` in Visual Basic) 문을 포함 하는 메서드는 제외의 후보입니다. 코드 베이스를 개발 주기에서 늦게 불안정 하거나 이전에 제공 된 코스에서 예기치 않은 런타임 동작이 발생 하는 위험이 코드를 리팩터링할 때의 유지 관리 이점 보다 클 수 있습니다.

## 순환 복잡성을 계산 하는 방법

순환 복잡성은 다음에 1을 더하여 계산 됩니다.

- 분기 수 (예: `if`, `while` 및 `do`)
- `case` 의 문 수 `switch`

# How to reduce CC?

- CC?

```
01. public void MethodDay(DayOfWeek day)
02.     {
03.
04.         switch (day)
05.         {
06.             case DayOfWeek.Monday:
07.                 Console.WriteLine("Today is Monday!");
08.                 break;
09.             case DayOfWeek.Tuesday:
10.                 Console.WriteLine("Today is Tuesday!");
11.                 break;
12.             case DayOfWeek.Wednesday:
13.                 Console.WriteLine("Today is Wednesday!");
14.                 break;
15.             case DayOfWeek.Thursday:
16.                 Console.WriteLine("Today is Thursday!");
17.                 break;
18.             case DayOfWeek.Friday:
19.                 Console.WriteLine("Today is Friday!");
20.                 break;
21.             case DayOfWeek.Saturday:
22.                 Console.WriteLine("Today is Saturday!");
23.                 break;
24.             case DayOfWeek.Sunday:
25.                 Console.WriteLine("Today is Sunday!");
26.                 break;
27.         }
28.     }
```

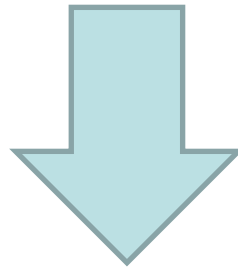
# How to reduce CC?

- CC?

```
01. public void MethodDayReduce(DayOfWeek day)
02.     {
03.         var factory = Activator.CreateInstance(Type.GetType($"
04.         {day.ToString()}DayFactory")) as IDayFactory;
05.         factory.Write();
06.     }
07. public interface IDayFactory
08.     {
09.         void Write();
10.     }
11.
12. public class MondayDayFactory : IDayFactory
13.     {
14.         public void Write()
15.         {
16.             Console.WriteLine("Today is Monday!");
17.         }
18.     }
```

# How to reduce CC?

```
01. public void Method(bool condition1, bool condition2)
02.     {
03.         if (condition1 || condition2 && (!condition1))
04.         {
05.             Console.WriteLine("Hello World!");
06.         }
07.     }
```



```
01. public void MethodReduce(bool condition1, bool condition2)
02.     {
03.         var negative = (!condition1);
04.         condition2 = condition2 && negative;
05.         condition2 = condition2 || condition1;
06.         if (condition2)
07.         {
08.             Console.WriteLine("Hello World!");
09.         }
10.     }
```

*Any fool can write code that a computer can understand. Good programmers write code that humans can understand*

*– Martin Fowler*

# Inter Modular Complexity

- Measures Dependencies Between Modules
  - Draw graph
    - Modules = nodes
    - Edges connecting modules may denote several relations, most often: A uses B (e.g., procedure call)

# The Uses Relation

- The *call-graph*
  - chaos (general directed graph)
  - hierarchy (acyclic graph)
  - strict hierarchy (layers)
  - tree



# In a picture:

