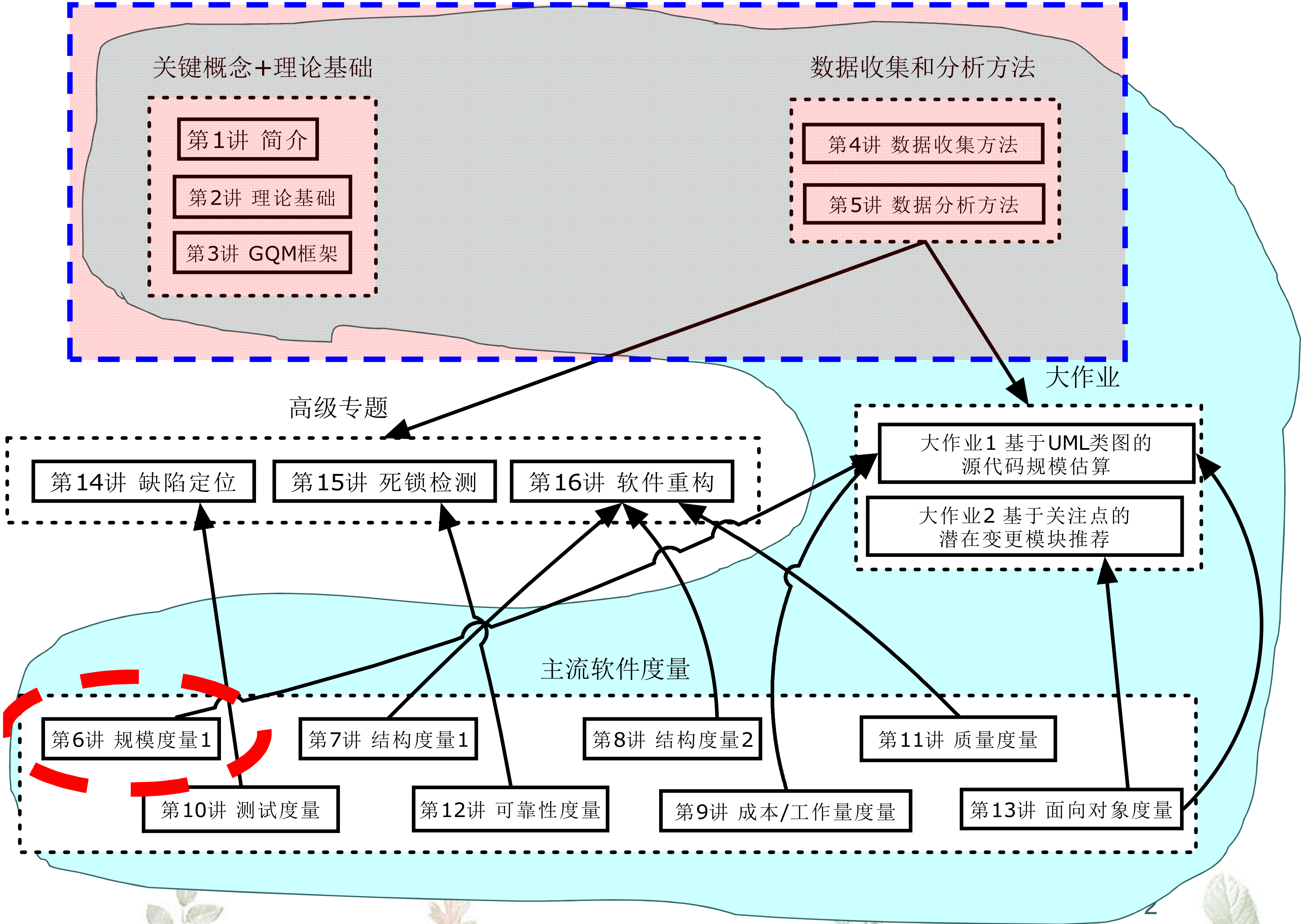
Software Metrics

Lecture 6

Size metrics

Yuming Zhou


关键概念+理论基础
第1讲 简介
第2讲 理论基础
第3讲 GQM框架
数据收集和分析方法
第4讲 数据收集方法
第5讲 数据分析方法
高级专题
第14讲 缺陷定位
第15讲 死锁检测
第16讲 软件重构
大作业
大作业1 基于UML类图的源代码规模估算
大作业2 基于关注点的潜在变更模块推荐
主流软件度量
第6讲 规模度量1
第7讲 结构度量1
第8讲 结构度量2
第11讲 质量度量
第10讲 测试度量
第12讲 可靠性度量
第9讲 成本/工作量度量
第13讲 面向对象度量

Software Size

- Size, an internal product attributes, which can be measured statically
- It is necessary to define software size in terms of more than one internal attributes
- Size measurement must reflect *effort*, *cost* and *productivity*

Contents

- Size: Length
- Size: Functionality

Section 1

Size: Length

- **Code**
- **Specification**
- **Design**

Software Size: Length

- Length is the "physical size" of the product

- In software development, there are three major development products: **specification**, **design**, and **code**

- The length of the specification can indicate how long the design is likely to be, which in turn is a predictor of code length

Length: Code - LOC /1

- The most commonly used metric LOC
  - *NCLOC:* non-commented source line of code or effective lines of code (ELOC)
  - *CLOC:* commented source line of code

- We can define:

  total length (LOC) = NCLOC + CLOC

- The ratio: *CLOC/LOC* measures the density of comments in a program

Length: Code - LOC /2

Variations of LOC(P249):

- Count of physical lines including blank lines
- Count of all lines except blank lines and comments
- Count of all statements except comments (statements taking more than one line count as only one line)
- Count of all lines except blank lines, comments, declarations and headings
- Count of only executable statements, not including exception conditions

Length: Code - LOC /3

- **Measurement Unit: Lines of Source Code**

Statement Type	*Includes*	*Excludes*
Executable	X	
Non-executable		
Declarations	X	
Compiler Directives	X	
Comments		X
On their own lines		X
On lines with source code		X
Banners and nonblank spacers		X
Blank (empty) comments		X
Blank Lines		X

Length: Code - LOC /4

- Advantages of LOC
  - Simple and automatically measurable
  - Correlates with programming effort (& cost)

- Disadvantage of LOC
  - Vague definition
  - Language dependability
  - Not available for early planning
  - Developers' skill dependability

① 存在最优的模块规模吗？

缺陷密度最小

K. El Emam et al. The optimal class size for object-oriented software. IEEE TSE, 2002, 28(5)

① 存在最优的模块规模吗？

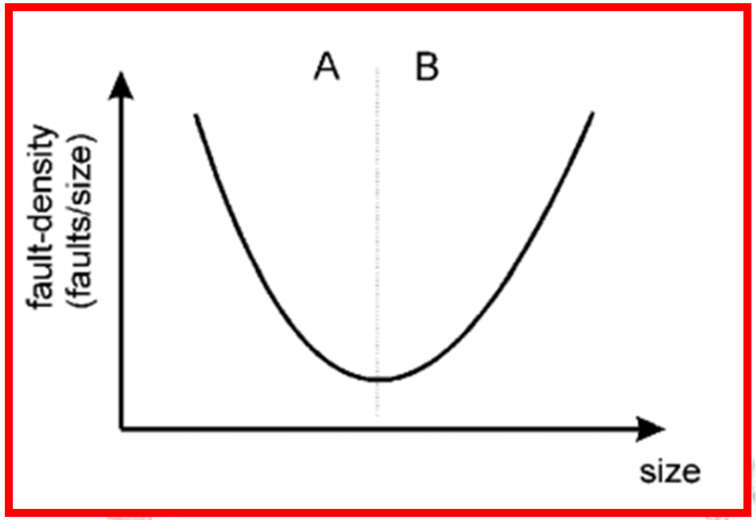
Ada package: 225 LOC

Columbus-Assembler: 400 LOC

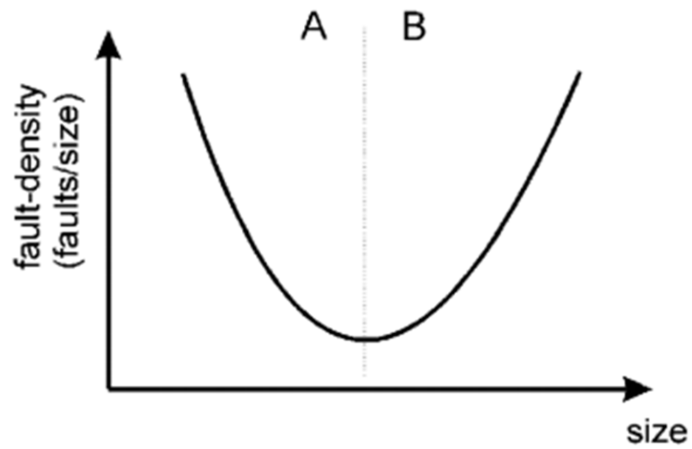
Jovial: 877 LOC

Pascal/Fortran: 100-150 LOC

…

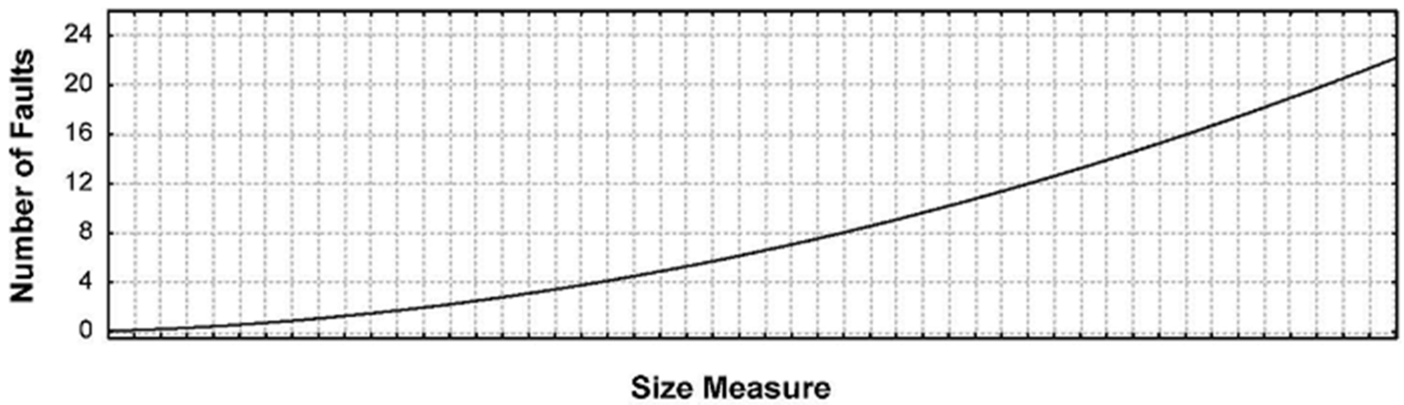


① 存在最优的模块规模吗？

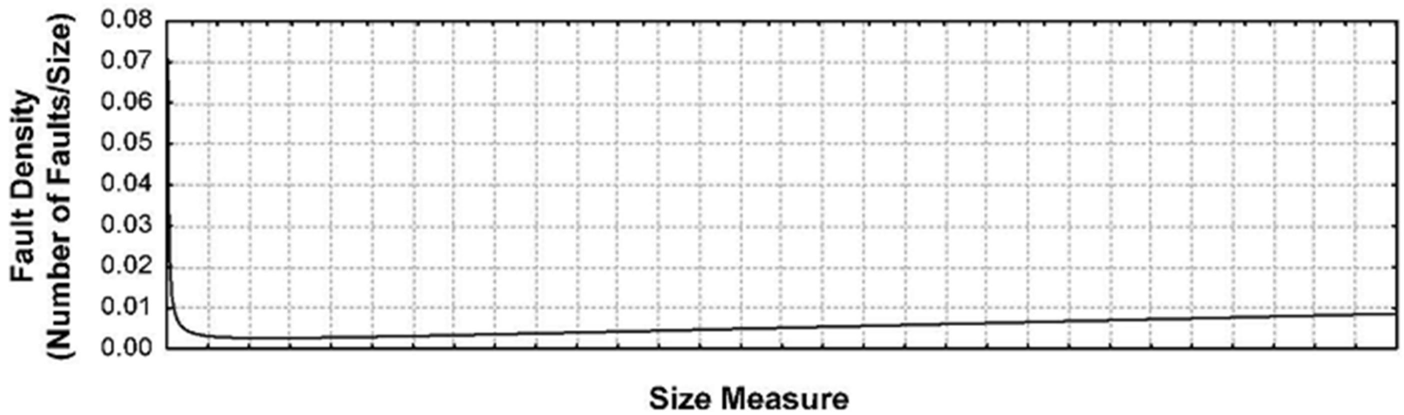


左半部原因：

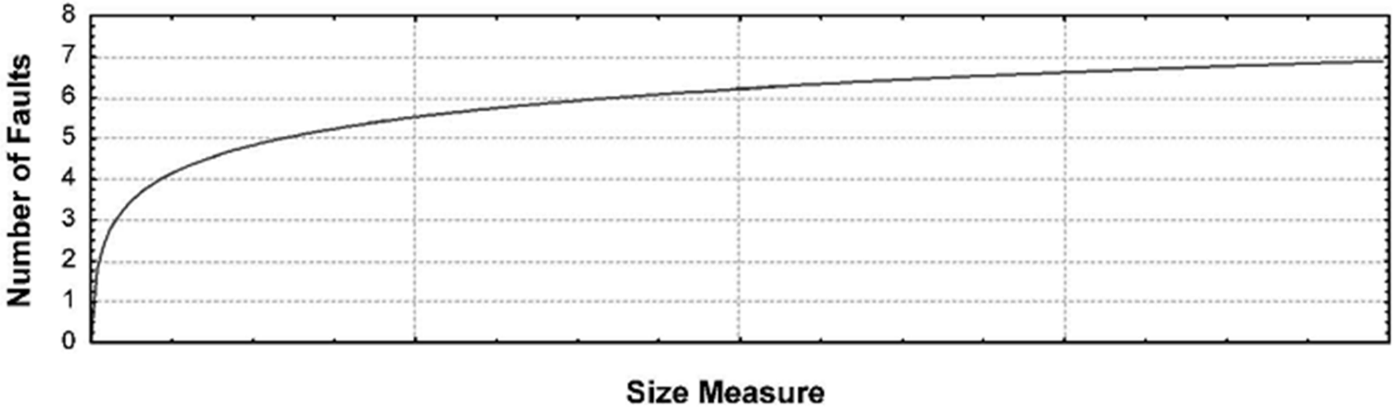
- 在规模大的模块中，有些部分没有被测试到
- 许多缺陷是接口缺陷，它们是均匀分布的。但用缺陷密度表示时，小模块的缺陷密度大
- Faults/loc 与 loc之间负相关



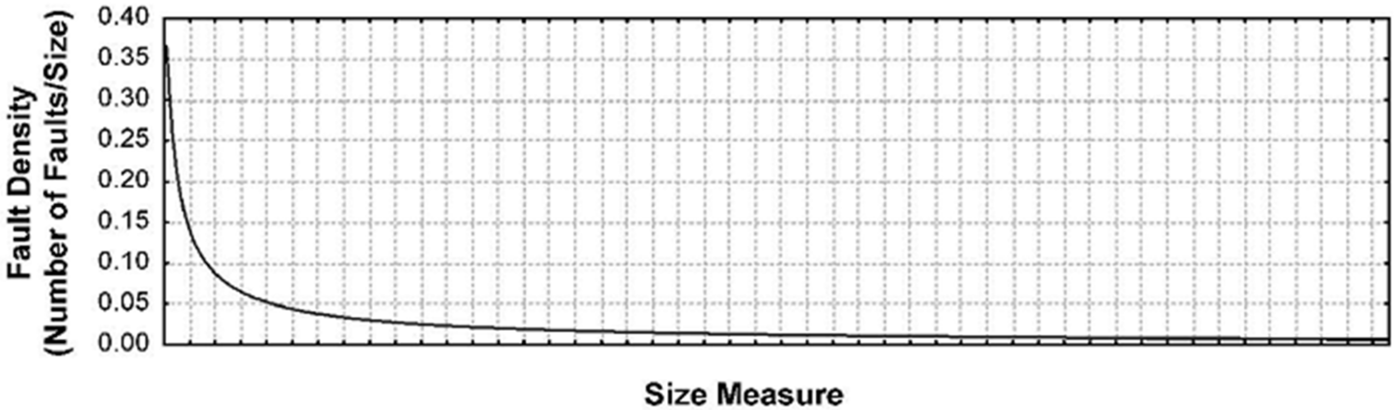
Number of Faults
24
20
16
12
8
4
0
Size Measure



Fault Density
(Number of Faults/Size)
0.08
0.07
0.06
0.05
0.04
0.03
0.02
0.01
0.00
Size Measure

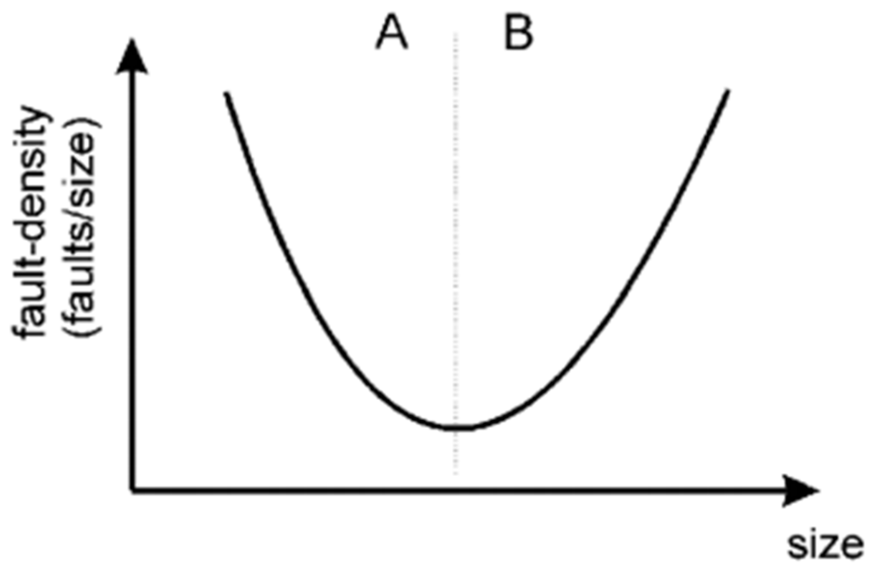


8
7
6
5
4
3
2
1
0
Number of Faults
Size Measure



0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
Fault Density
(Number of Faults/Size)
Size Measure

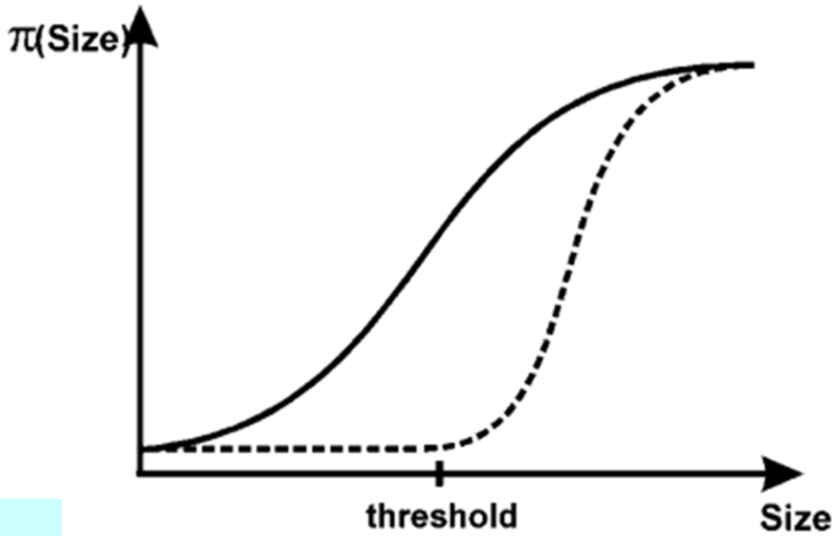
① 存在最优的模块规模吗？



右半部：阈值效果？

① 存在最优的模块规模吗？

$$\pi = \frac{1}{1+e^{-(\beta_0+\beta_1 size)}}$$



$$\pi = \frac{1}{1+e^{-(\beta_0+\beta_1(size-\tau)I_+(size-\tau))}}$$

$$I_+(z) = \begin{cases} 0 & if\ z \le 0 \\ 1 & if\ z > 0 \end{cases}$$

$$H_0 : \tau \le size^{(1)} = \min size,$$

① 存在最优的模块规模吗？

TABLE 2
Threshold and No-Threshold Model Results and Their Comparison for the Three Systems and Four Size Measures

System	Size Measure	No Threshold Model			Threshold Model			Comparison of Models	
		R^2	β_1 Coefficient (s.e.)	p-value	R^2	β_1 Coefficient (s.e.)	p-value	Estimated Threshold Value	Chi-Square (p-value)
C++ System 1	STMT	0.040	0.001915 (0.001261)	0.036	0.061	0.1061 (0.3743)	0.01	671	2.256 (0.133)
C++ System 2	SLOC	0.0578	0.01075 (0.003274)	0.00022	0.0578	0.01075 (0.003274)	0.00022	3	--[28]
Java System	NM	0.07	0.0571 (0.02464)	0.0103	0.07	0.0571 (0.02464)	0.0103	1	--[29]
	NAI	0.037	0.04347 (0.02423)	0.0631	0.0416	7.3956 (31.1326)	0.0499	39[30]	0.39 (0.53)

For all models the condition number was found to be always less than three, indicating that collinearity was not a problem. 28 For the threshol

理论上不存在阈值

不存在最优的规模

② 规模的阈值是多少？

背景

在实际的软件开发活动中，人们需要知道OO度量的阈值，以便识别一个面向对象系统中可能“不稳定”的类，从而能够有针对性地进行质量保证活动

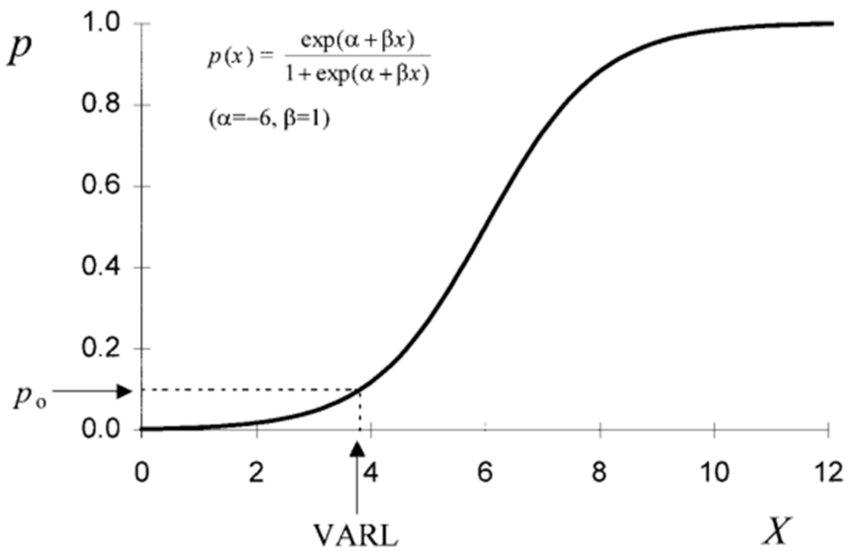
问题描述：规模的阈值是多少？

R. Shatnawi. A quantitative investigation of the acceptable risk levels of object-oriented metrics in open-source systems. IEEE TSE, 2010, 36(2)

② 规模的阈值是多少？

阈值模型

$$p(x)=\frac{e^{\alpha+\beta x}}{1+e^{\alpha+\beta x}}$$



可接受风险值VARL与风险级别p_0之间的关系

$$VARL(p_0)=p^{-1}(p_0)=\frac{1}{\beta}\left(\log\left(\frac{p_0}{1-p_0}\right)-\alpha\right)$$

对一个类而言，只要它的*X*值小于*VARL(p0)*，那么“不稳定”的概率就小于*p0*。

② 规模的阈值是多少？

其他阈值生成方法

1. **平均值法**。利用训练数据集上OO度量的平均值作为阈值(先计算单个系统上的平均值，然后利用元分析方法得到总体上的平均值)

2. **平均值±标准差法**。利用平均值的一标准差范围作为阈值：如果OO度量与不稳定正相关，那么其阈值为“平均值+标准差”(即Mean+SD)；反之为“平均值–标准差”(Mean–SD)

3. **风险剖面法**。
   ① 低风险阈值LR70：大于该阈值的类的代码量占训练集中系统代码总量的30%
   ② 中等风险阈值MR80：大于该阈值的类的代码量占训练集中系统代码总量的20%
   ③ 高风险阈值HR90：大于该阈值的类的代码量占训练集中系统代码总量的10%

② 规模的阈值是多少？

度量	logistic模型		平均值法		平均值 ± 标准差法			风险剖面法					
	阈值	g-mean	阈值	g-mean	类型	阈值	g-mean	LR70	g-mean	MR80	g-mean	HR90	g-mean
SLOC	64	0.669	111	0.625	Mean+SD	145	0.589	388	0.343	580	0.238	992	0.125
Stmts	42	0.669	74	0.625	Mean+SD	96	0.591	263	0.345	401	0.229	694	0.122
NumPara	7	0.635	11	0.595	Mean+SD	13	0.567	28	0.379	42	0.275	75	0.148
NMIMP	6	0.628	10	0.601	Mean+SD	12	0.569	24	0.377	34	0.263	54	0.150
NAIMP	3	0.606	5	0.576	Mean+SD	6	0.551	11	0.420	18	0.291	31	0.170
NM	12	0.544	20	0.583	Mean+SD	26	0.543	35	0.491	50	0.413	84	0.297
NA	5	0.606	9	0.562	Mean+SD	12	0.517	17	0.450	25	0.373	41	0.269

	预测情况	
实际情况	稳定的 (m ≥ t)	不稳定的 (m < t)
稳定的	TP	FN
不稳定的	FP	TN

$$TNR = \frac{TN}{FP + TN}$$

$$TPR = \frac{TP}{TP + FN}$$

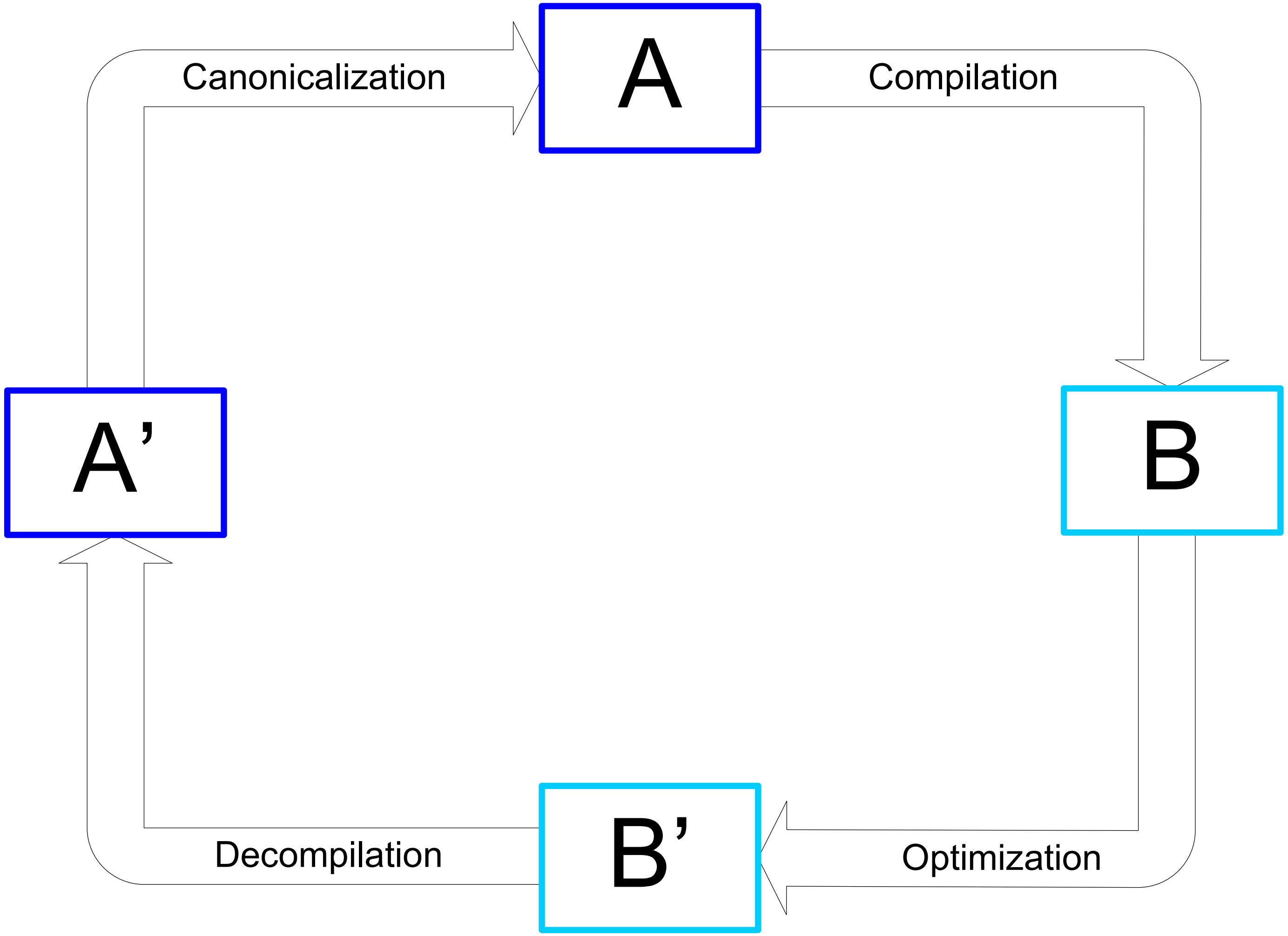
$$gmean = \sqrt{TPR \times TNR}$$

③ 规模与缺陷间存在什么关系？

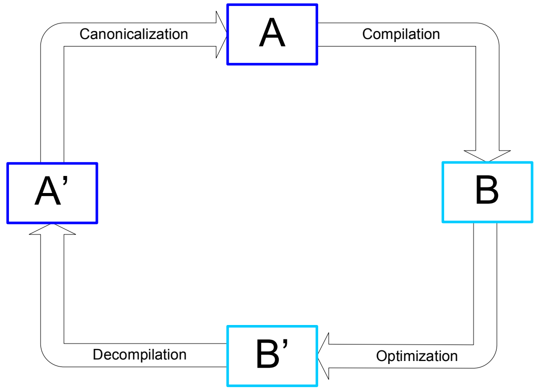
A. Koru et al. An investigation into the functional form of the size-defect relationship for software modules. IEEE TSE, 2009, 35(2)

M.D. Syer, M. Nagappan, B. Adams, A.E. Hassan. Replicating and re-evaluating the theory of relative defect-proneness. IEEE TSE, 2014, accepted.

Halstead's View of Programming

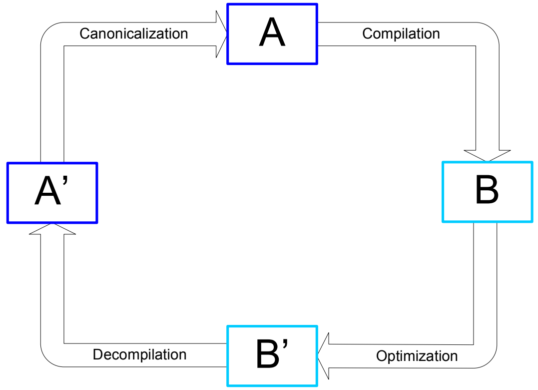


Halstead's View



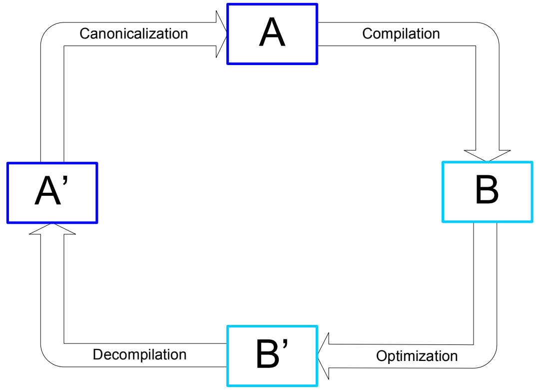
- Compilation
  - Compiled
  - Translated from a higher to lower-level language (including assembly and machine language)

Halstead's View



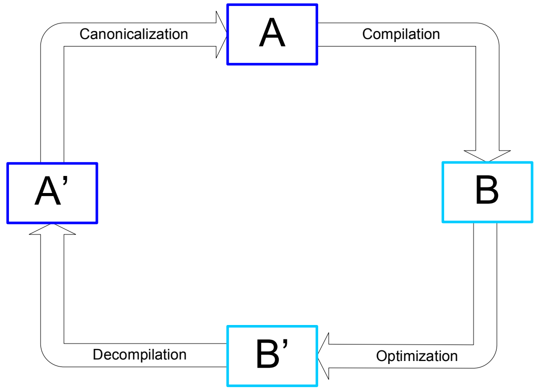
- Optimization
  - Making the object or executable "more efficient" with respect to a given machine language

Halstead's View



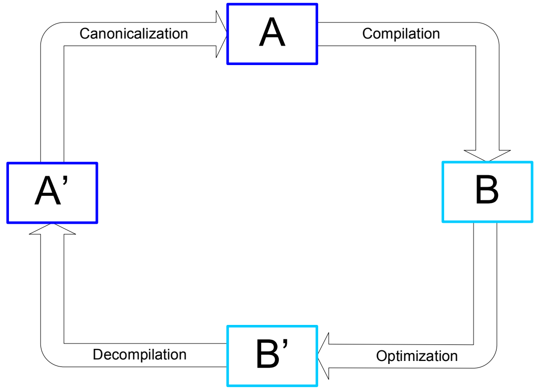
- Decompilation
  - Also known as "inverse compilation"
  - Translation from a lower-level language to a higher-level language

Halstead's View



- Canonicalization
  - Change by such things as:
    - Unwinding loops
    - Bringing procedures in line to replace calls
    - Reversing optimization
    - Etc.
  - The canonical form can be pretty much whatever you wish to define it as.

Halstead's View



- An algorithm can be processed through this cycle without loss of information
- Problem: Optimization and Inverse Compilation cannot be both rigorous and practical at the same time
  - Halstead's model is the "ideal" – like the Carnot cycle in thermodynamics

Halstead's View

- An algorithm consists of
  - Operators
  - Operands
  - Nothing else

Basic Measurable Properties of Algorithms

- μ_1 = number of unique or distinct *operators* appearing in the implementation
- μ_2 = number of unique or distinct *operands* appearing in that implementation
- N_1 = total usage of all of the operators appearing in that implementation
- N_2 = total usage of all of the operands appearing in that implementation
- $f_{1,j}$ = number of occurrences of the jth most frequently occurring operator, where $j = 1, 2, \ldots, \mu_1$
- $f_{2,j}$ = number of occurrences of the jth most frequently occurring operands, where $j = 1, 2, \ldots, \mu_2$

Some Terminology

- Vocabulary (μ): how many symbols are in the implementation
  - $\mu = \mu_1 + \mu_2$
  - Vocabulary = # of operators + # of operands
- Implementation Length
  - $N = N_1 + N_2$
  - Length = total usage of operators + total usage of operands

Basic Relationships

$$N_1 = \sum_{j=1}^{j=\mu_1} f_{1,j}$$

$$N_2 = \sum_{j=1}^{j=\mu_2} f_{2,j}$$

$$N = \sum_{j=1}^{j=\mu_1} f_{i,j} + \sum_{j=1}^{j=\mu_2} f_{2,j} = \sum_{i=1}^{i=2} \sum_{j=1}^{j=\mu_i} f_{i,j}$$

Euclid's Algorithm (Clausen)

```
          IF (A=0)
LAST:     BEGIN GCD := B; RETURN END;
          IF (B=0)
          BEGIN GCD := A; RETURN END;
HERE:     G := A/B; R:=A-B×G;
          IF (R=0) GO TO LAST;
          A := B; B:=R; GO TO HERE
```

Euclid's Algorithm's Basic Metrics

Operator	j	$f_{1,j}$
;	1	9
:=	2	6
() or BEGIN..END	3	5
IF	4	3
=	5	3
/	6	1
-	7	1
×	8	1
GOTO HERE	9	1
GOTO LAST	10	1

Operand	j	$f_{2,j}$
B	1	6
A	2	5
O	3	3
R	4	3
G	5	2
GCD	6	2

$$\mu_1 = 10 \qquad N_1 = 31$$

$$\mu_2 = 6 \qquad N_2 = 21$$

Another Example

- For the following C program:

```
#include<stdio.h>
main()
{
int a ;
scanf (“%d”, &a);
if ( a >= 10 )
   if ( a < 20 )    printf ("10 < a< 20 %d\n" , a);
   else             printf ("a >= 20    %d\n" , a);
else                printf ("a <= 10    %d\n" , a);
}
```

Example (cont'd)

- Determine the number of distinct operators (μ_1)
- Determine the number of distinct operands (μ_2)
- Determine the program length in terms of the total number of occurrences of operators (N_1) and operands (N_2), using the relation: $N = N_1 + N_2$

Example (cont'd)

Operators	Number of occurrences	Operators	Number of occurrences
#	1	<=	1
include	1	\n	3
stdio.h	1	printf	3
< … >	1	<	3
main	1	>=	2
(…)	7	if … else	2
{ … }	1	&	1
int	1	,	4
;	5	%d	4
scanf	1	“ … “	4

Example (cont'd)

Operands	Number of occurrences
a	10
10	3
20	3
$\mu_2 = 3$	$N_2 = 16$
$\mu_1 = 20$	$N_1 = 47$
$N = N_1 + N_2 = 63$	

Exercise

```
procedure BubbleSort(var a : IntArray;
                         N : Positive);
var
   j, t  : integer;
begin
   repeat
      t := a(1);
      for j := 2 to N do
         if a(j–1) > a(j) then
         begin
            t := a(j–1);
            a(j–1) := a (j);
            a(j) := t;
         end
   until t = a(1)
end;
```

Exercise

Operator	Count
;	4
(. . .) (array subscript)	8
:=	4
-	3
$>$	1
=	1
if . . . then	1
repeat . . . until	1
for . . . := . . . to . . . do	1
begin . . . end	1
procedure . . . ;	1
$\eta_1 = 11$	$N_1 = 26$

Exercise

Operand	Count
j	7
t	4
a	8
N	1
1	5
2	1
$\eta_2 = 6$	$N_2 = 26$

Exercise

Length:

$$
\begin{aligned}
N &= N_1 + N_2 \\
&= 26 + 26 \\
&= 52
\end{aligned}
$$

Estimated Length:

$$
\begin{aligned}
\hat{N} &= \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2 \\
&= 11 \times \log_2 11 + 6 \times \log_2 6 \\
&= 53.56
\end{aligned}
$$

Halstead Length

- N = N1 + N2

- An approximation formula has also been derived that estimates the length of an algorithm from the size of the vocabulary

$$\hat{N} = \mu_1 \log_2 \mu_1 + \mu_2 \log_2 \mu_2$$

Halstead's experiment

14 algorithms

No.	N	$\hat{N}$	$\hat{N} - N$
1	104	104	0
2	82	77	5
3	453	300	153
4	132	139	7
5	123	123	0
6	98	101	3
7	59	62	3

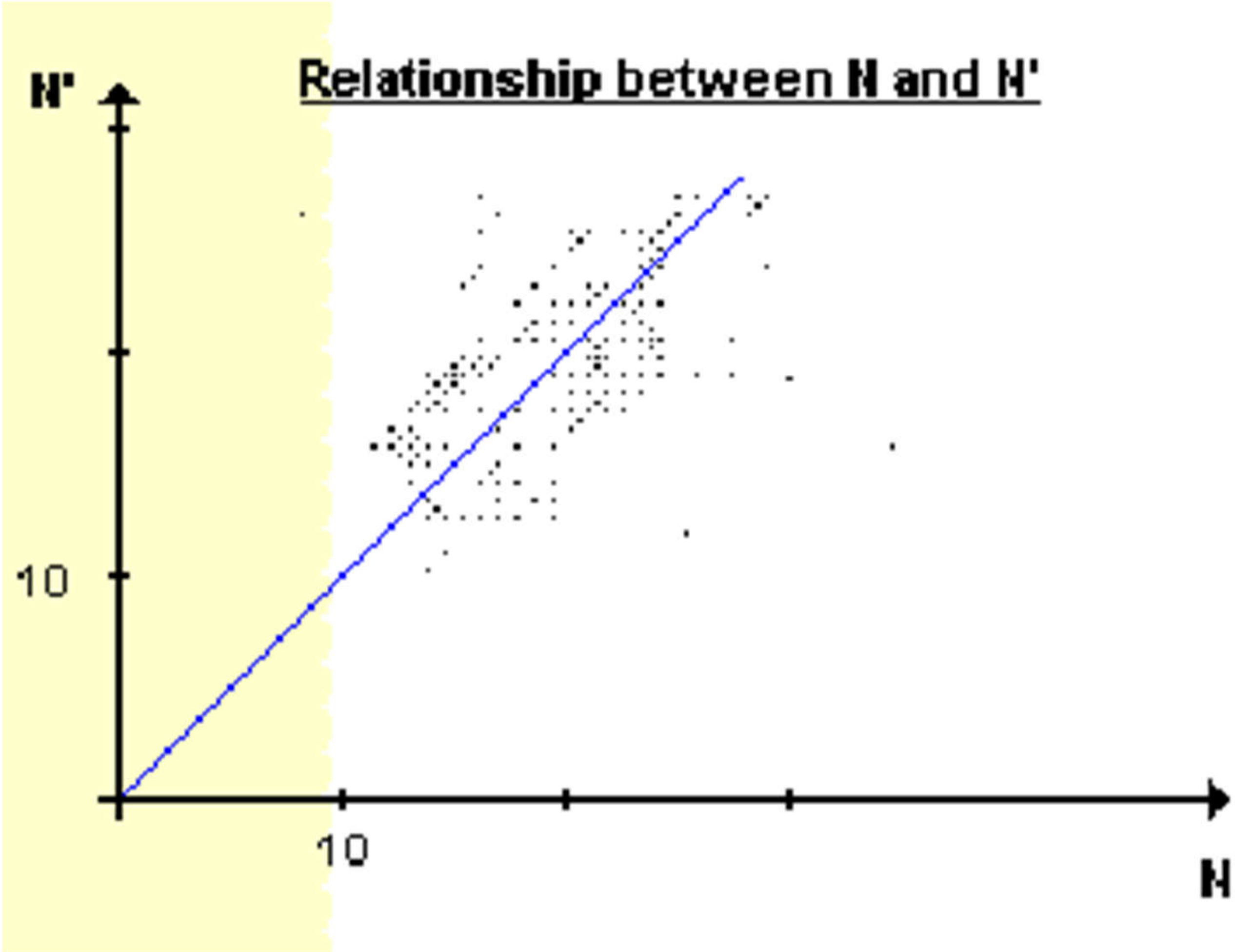
No.	N	$\hat{N}$	$\hat{N} - N$
8	131	117	14
9	314	288	26
10	46	52	6
11	53	52	1
12	59	62	3
13	59	57	2
14	186	163	23

James Elshoff's experiment

120 PL/I programs (>100,000 statements)

Class i	Program # In Class	Mean Values	
		N	Ñ
14	3	18592	10091
13	17	10685	11049
12	23	5751	6005
11	39	3165	3318
10	17	1590	1663
9	11	831	911
8	4	369	522
7	5	198	195
6	1	122	129
totals	120	41303	42883

The relationship between N and Ñ



Program Volume

- Observe that there is
  - An absolute minimum length for representation of the longest operator or operand name if expressed in bits.
  - If the number of elements in a vocabulary is X, then the minimum length is $\log_2 x$
- Thus, $V = N \log_2 \mu$
  - The volume of the algorithm is the number of elements times the encoded length of a single element

Program Volume

$V = N \log_2 \mu$

- 用二进制编码时的程序长度
- 写长度为N的程序时，脑袋中比较的次数

Potential Volume

- 同一个算法可以有多个不同的实现
- 在这些实现中，“最短长度”称为“potential volume”

$$\boxed{V^* = (2+\mu_2^*)\log_2(2+\mu_2^*)}$$

Potential Volume

$$V = N \log_2 \mu$$

$$where\ \ N = N_1 + N_2$$

$$and \quad N_1^* = 2 \quad and \quad N_2^* = \mu_2^*$$

$$V^* = (2 + \mu_2^*) \log_2 (2 + \mu_2^*)$$

NOTE: minimum operators consist of operation itself and grouping operator
minimum operands consist of inputs and outputs

Euclid's Algorithm Volume

- $V = (N_1 + N_2) \log_2 (\mu_1 + \mu_2)$
  $= (31 + 21) \log_2 (10 + 6)$
  $= 208$
- $V^* = (2 + \mu_2^*) \log_2 (2 + \mu_2^*)$
  $= (2 + 3) \log_2 (2 + 3)$
  $= 11.6$

Program Level

- What is the abstract “level” of a particular implementation of an algorithm?
- $L = V^* / V$
  - Note that “highest” level is 1

- Program volume (V) is probably the best measurement of complexity
- Higher-level languages generate lower volume for a given algorithm

Language difficulty

$$D = 1/L = V/V^*$$

programming practices such as the redundant usage of operands, or the failure to use higher level control constructs will tend to increase the Volume as well as the Difficulty

Language difficulty

$$\hat{D} = \frac{\mu_1}{2} \times \frac{N_2}{\mu_2}$$

实践中，使用上述估算公式

(1) 附加的运算符越多，编写程序的难度越大

(2) 操作数重复次数越多，编写程序的难度越大

Program effort

- Size increases, effort increases
- Higher difficulty (lower level), larger effort

$$E = V / L = V \times D$$

The unit of measurement of E is "elementary mental discriminations".

Programming time

T = *E / S*

- 心理学：每秒可以做5～20次"elementary mental discriminations"
- 实际取18

Programming time

Sorting Experiment Results

Program Number	Actual Time (minutes)	Est. Time (minutes)
1	6	7
2	12	6
3	13	10
4	14	14
5	15	15
6	95	44
7	127	164
8	173	174

Critics of Halstead's work

- The treatment of basic and derived metrics is somehow confusing
- Unable to be extended to include the size for specification and design

M. Halstead. Elements of Software Science, Elsevier North-Holland, New, York, N.Y" (1977).

V. Shen et al. Software science revisited: a critical analysis of the theory and its empirical support. Technical Report, 1981

S. Lessmann et al. Benchmarking classification models for software defect prediction: a proposed framework and novel findings. IEEE TSE, 2008, 34(4)

T. Menzies et al. Data mining static code attributes to learn defect predictors. IEEE TSE, 2007, 33(1)

Section 2

Size: Functionality

- Function Point
- Feature Point
- Object Point
- Use-case Point

Function-Oriented Metrics

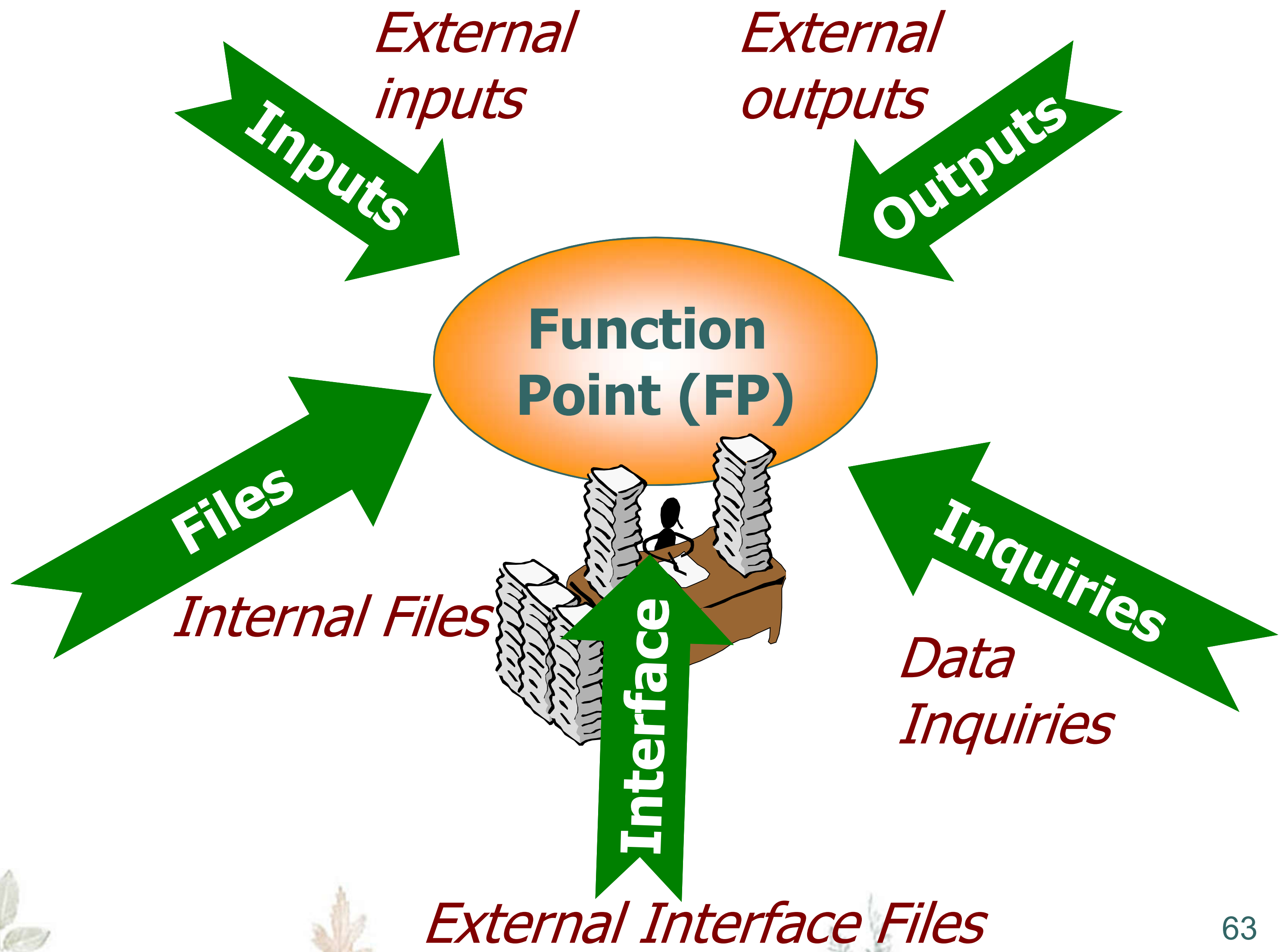
The idea is that a product with more functionality will be larger in size

- Albrecht (1979) who suggested a productivity measurement approach called the **Function Point (FP)** method

- Measure the amount of functionality in a system based upon the *system specification*

Estimation before implementation!

Function Point (FP) /1



Function Point (FP) /2

- FP is computed in two steps:
  1) Calculating an ***Unadjusted Function point Count (UFC)***
  2) Multiplying the UFC by a ***Technical Complexity Factor (TCF)***

- The final (adjusted) Function Point is:

$$FP = UFC \times TCF$$

Function Point (FP) /3

- Counts are made for the following categories:
  - ***Number of external inputs (N_i):*** data, control information (such as file names and menu selections), change the status of its internal logical file(s)
  - ***Number of external outputs (N_o):*** data or control information produced by the software systems, e.g., reports and messages

Function Point (FP) /4

- Counts are made for the following categories:
  - ***Number of external inquiries*** **(N_q):** input/output combinations, without changing any status of internal logical files
  - ***Number of external interface files*** **(N_{ef})**
  - ***Number of internal files*** **(N_{if})**

Unadjusted FP Count (UFC)

- **function point complexity weights**

Item	Weighting Factor		
	Simple	Average	Complex
External inputs (N_i)	3	4	6
External outputs (N_o)	4	5	7
External inquiries (N_q)	3	4	6
External interface files (N_{ef})	7	10	15
Internal files (N_{if})	5	7	10

$$UFC = 4N_i + 5N_o + 4N_q + 10N_{ef} + 7N_{if}$$

Tech. Complexity Factor (TCF) /1

- Technical Complexity Factor (*TCF*)

F1	Reliable back-up and recovery	F2	Data communications
F3	Distributed functions	F4	Performance
F5	Heavily used configuration	F6	Online data entry
F7	Operational ease	F8	Online update
F9	Complex interface	F10	Complex processing
F11	Reusability	F12	Installation ease
F13	Multiple sites	F14	Facilitate change

Tech. Complexity Factor (TCF) /2

- Each component is rated from 0 to 5

- The TCF can then be calculated as

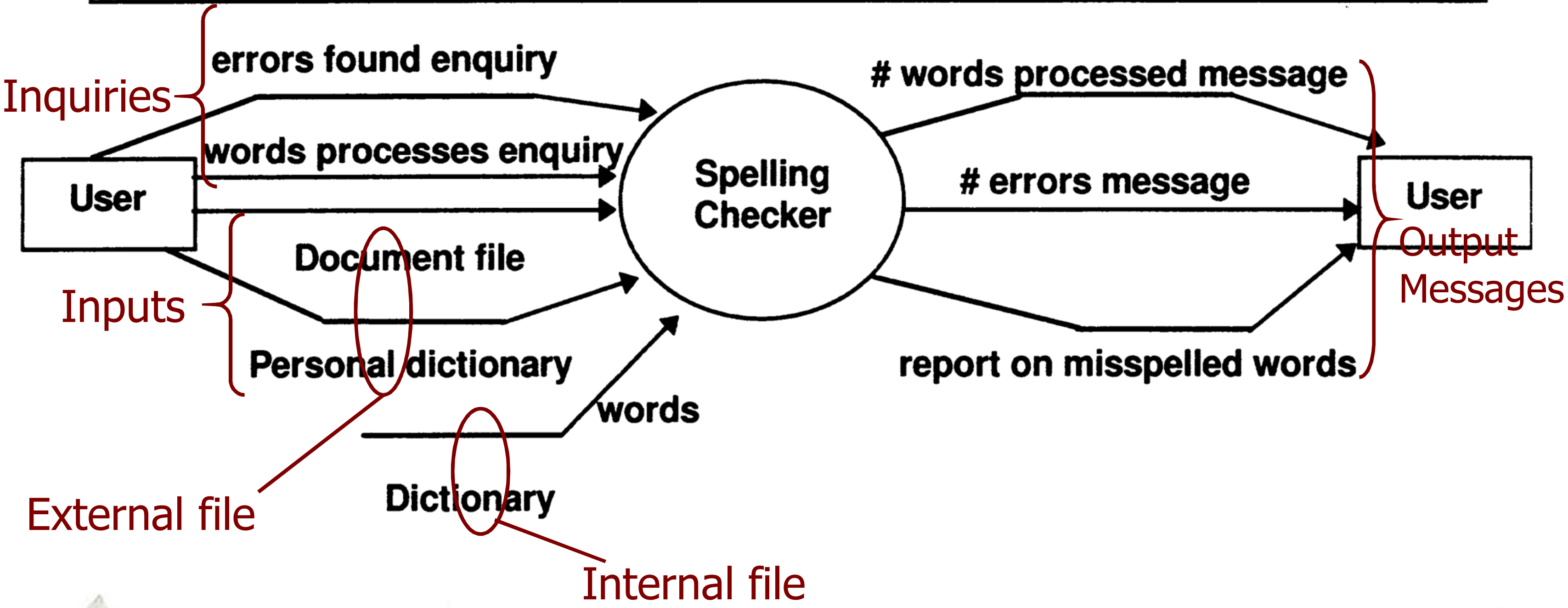
$$TCF = 0.65 + 0.01\sum_{j=1}^{14} F_j$$

- The TCF varies from 0.65 (if all F_j are set to 0) to 1.35 (if all F_j are set to 5)

$$FP = UFC \times TCF$$

FP: Example

Spell-Checker Spec: The checker accepts as input a document file and an optional personal dictionary file. The checker lists all words not contained in either of these files. The user can query the number of words processed and the number of spelling errors found at any stage during processing.



FP: Example (cont'd)

- $N_i = 2$: document filename, personal dictionary-name
- $N_o = 3$: misspelled word report, number-of-words-processed message, number-of-errors-so-far message
- $N_q = 2$: words processed, errors so far
- $N_{ef} = 2$: document file, personal dictionary
- $N_{if} = 1$: dictionary

$$\mathbf{UFC = 4 \times 2 + 5 \times 3 + 4 \times 2 + 10 \times 2 + 7 \times 1 = 58}$$

- Suppose that

  F1=F2=F6=F7=F8=F14=3; F4=F10=5

  TCF = 0.65 + 0.01(18+10) + 0.93

  $$\mathbf{FP = 58 \times 0.93 \approx 54}$$

FP vs. LOC /1

- A number of studies have attempted to relate LOC and FP metrics (Jones, 1996)
- average number of source code statements per function point

FP vs. LOC /2

Language	Level	Min	Average	Max
Machine language	0.10		640	
Assembly	1.00	237	320	416
C	2.50	60	128	170
Pascal	4.00		90	
C++	6.00	40	55	140
Visual C++	9.50		34	
PowerBuilder	20.00		16	
Excel	57.00		5.5	
Code generators	--		15	

FP vs. LOC/3(The Paradox of LOC)

Activity	Case A Assembler Version (10,000 Lines)	Case B Fortran Version (3,000 Lines)	Difference
Requirement	2 Months	2 Months	0
Design	3 Months	3 Months	0
Coding	10 Months	3 Months	-7
Integration/Test	5 Months	3 Months	-2
User Documentation	2 Months	2 Months	0
Management/Support	3 Months	2 Months	-1
Total	25 Months	15 Months	-10
Total Costs	$125,000	$75,000	($50,000)
Cost Per Source Line	$12.50	$25.00	$12.50
Lines Per Person Month	400	200	-200

FP vs. LOC/4(The Economic Validity of FP)

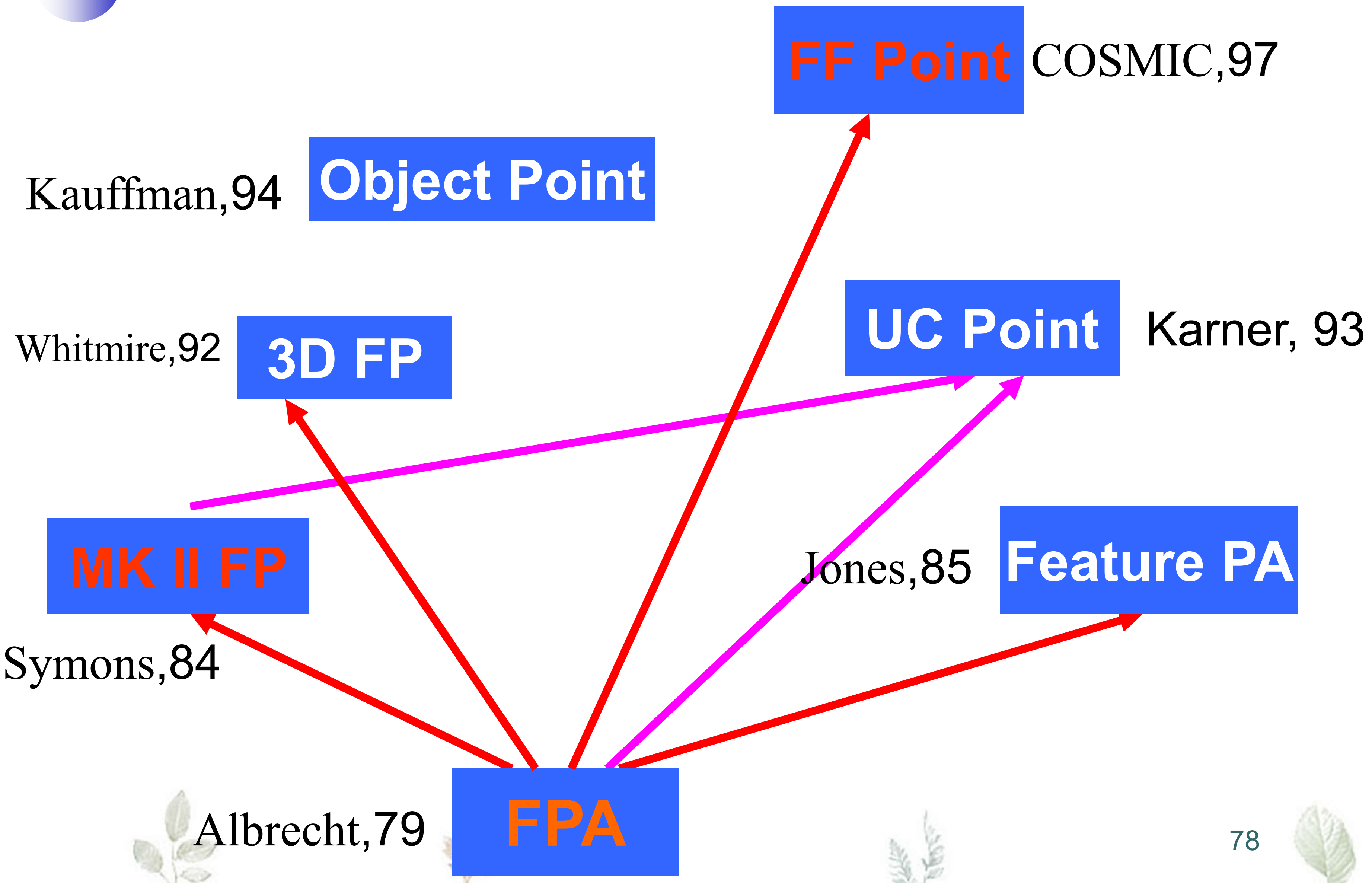
Activity	Case A Asssembler Version (30 F.P.)	Case B Fortran Version (30 F.P.)	Difference
Requirements	2 Months	2 Months	0
Design	3 Months	3 Months	0
Coding	10 Months	3 Months	-7
Integration/Test	5 Months	3 Months	-2
User Documentation	2 Months	2 Months	0
Management/Support	3 Months	2 Months	-1
Total	25 Months	15 Months	-10
Total Costs	$125,000	$75,000	($50,000)
Cost Per F.P.	$4,166.67	$2,500.00	($1,666.67)
F.P. Per Person Month	1.2	2	+ 0.8

FP: Critics

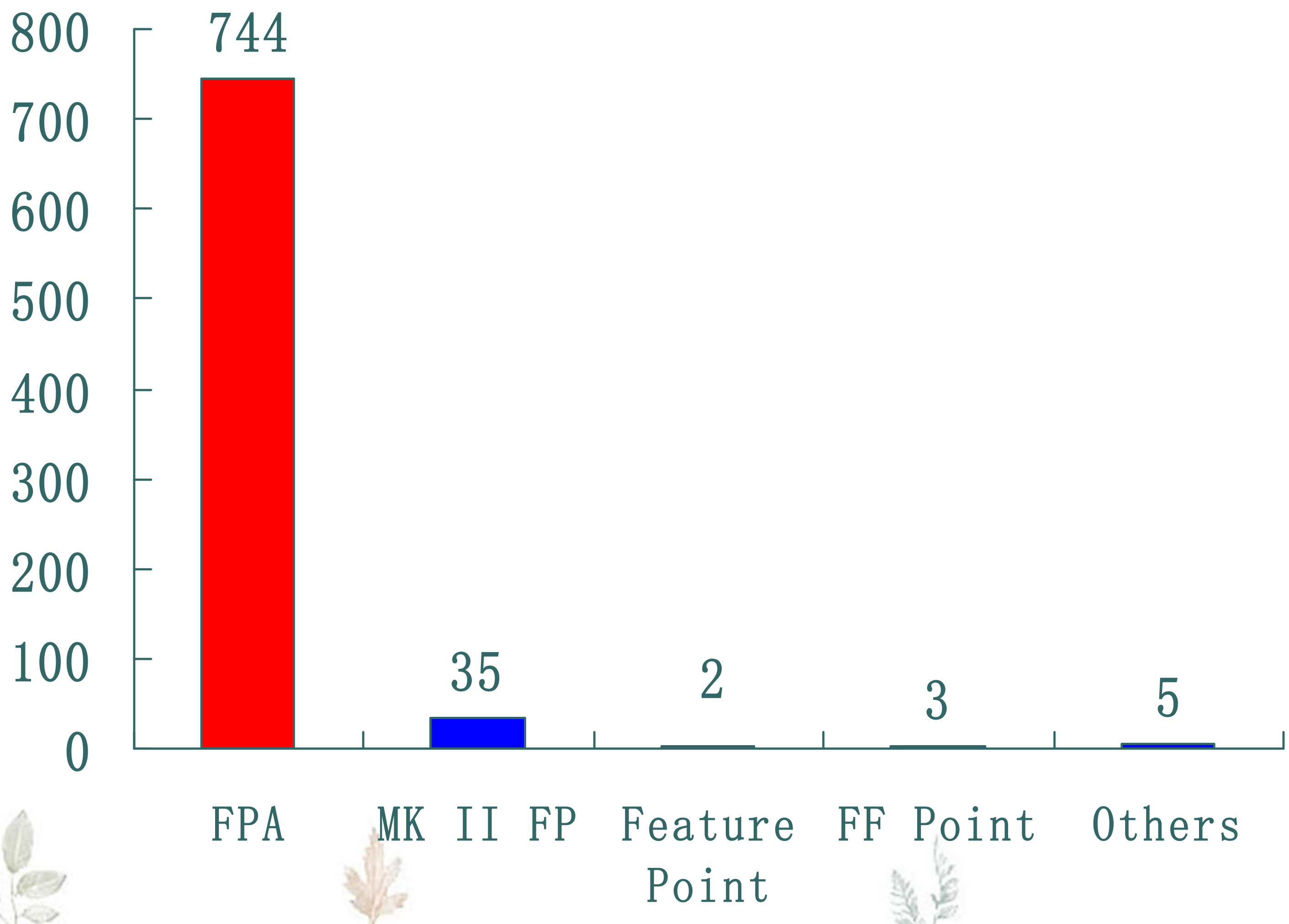
- FP is a subjective metric
- Function point calculation requires a full software system specification
- Not suitable for "complex" software, e.g., real-time and embedded applications

- Function Point Analysis: in 1979, Allan Albrecht (IBM) (IFPUG) **MIS**
- MK II FPA: in 1984, Charles Symons, number of complexity factors, a series of environmental factors and a different sets of weights (ISO/IEC 20968:2002) **MIS**
- Feature Point Analysis: in 1985, Capers Jones, extending the FPs counting to real-time and TLC environments **MIS&RT& SC**
- 3D Function Points: in 1992, Scott Whitmire (Boeing), data, controls and functions **RT& SC**
- Use Case Point: in 1993, Gustav Karner
- Object Point: in 1994, Kauffman, et. al. predict the functionality of a software product earlier in the life cycle than function points
- Full Function Points: in 1997, COSMIC, mainly oriented towards real-time and embedded systems, can be adopted in a general-purpose contex **MIS&RT**

Relations among FPs



On the fall of 1999, data about 789 projects had been collected



Feature Point /1

- "algorithmic complexity" is high such as real-time, process control, and embedded software applications

- For MIS, functions and feature points produce similar results. For RT, feature points produce counts about %20~%35 higher than function points

Feature Point /2

- Counts are made for the five FP categories, i.e., number of external inputs, external outputs, inquiries, internal files, external interfaces, plus:
  - **Algorithm (N_a):** A bounded computational problem such as inverting a matrix, decoding a bit string, or handling an interrupt

Feature Point /3

- Feature points are calculated using:
  - Number of external inputs ×4
  - Number of external outputs ×5
  - Number of external inquiries ×4
  - Number of external interface files ×7
  - Number of internal files ×7
  - Algorithms ×3

$$UF_eC = 4N_i + 5N_o + 4N_q + 7N_{ef} + 7N_{if} + 3N_a$$

- The UF_eC used in the function point calculation to calculate the feature points

Object Point /1

- Object points are used as an initial metric for size early in the development cycle

- An initial size metric is determined by counting the number of ***screens***, ***reports***, and ***components*** that will be used

- Each object is classified as ***simple***, ***medium***, or ***difficult***

Object Point /2

- **Object point complexity levels for screens**

Number of views contained	**Number and source of data tables**		
	Total <4 <2 servers <2 clients	**Total <8 2- 3 servers 3-5 clients**	**Total 8+ >3 servers >5 clients**
< 3	Simple	Simple	Medium
3-7	Simple	Medium	Difficult
8+	Medium	Difficult	Difficult

Object Point /3

- **Object point complexity levels for reports**

Number of views contained	Number and source of data tables		
	Total <4 <2 servers <2 clients	Total <8 2-3 servers 3-5 clients	Total 8+ >3 servers >5 clients
0 - 1	Simple	Simple	Medium
2 - 3	Simple	Medium	Difficult
4+	Medium	Difficult	Difficult

Object Point /4

- **Complexity level for object point**

Object Type	Simple	Medium	Difficult
Screen	1	2	3
Report	2	5	8
Component	-	-	10

New object points = (object points) x *(100 – r) / 100*

Assuming that *% r* of the objects will be reused from previous projects

Example: Object Point

- Suppose that you have
  - 4 screens: 2 simple (weight 1) and 1 medium (weight 2)
  - 3 reports: 2 simple (weight 2) and 1 medium (weight 5)

  then the total number of OP is:

$$\mathbf{OP = 2\times1 + 1\times2 + 2\times2 + 1\times5 = 13}$$

- If you have any acquired component, give it the weight 10 and add it to the OP

- define a percentage of reuse (say 10%) and then adjust the value of OP

$$\mathbf{OPnew = 13 \times (100\text{-}10)/100 = 11.7}$$

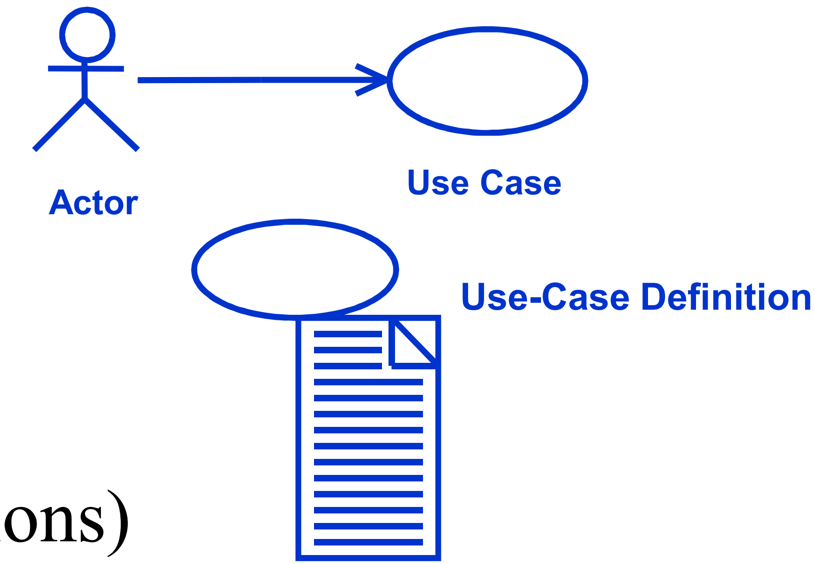
Use-Case Point /1

- Use-Case is a method to develop *requirements*

- **Question:** How to use Use-Cases to measure function point?

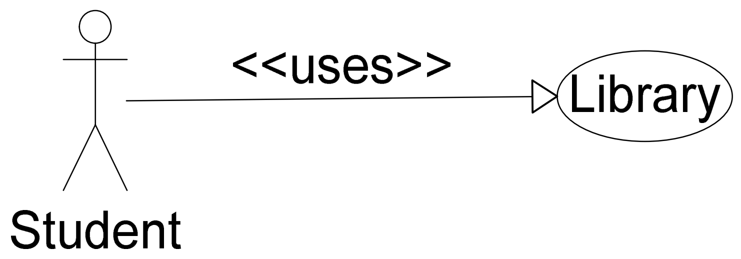
Use-Case Point /2

- A typical ***use-case definition*** document consists of:
  - Use Case name
  - Actor name
  - Objective
  - Preconditions
  - Results (Post-conditions)
  - Detailed description (actions & responses)
  - Exceptions and alternative courses



Use-Case Point /3

- We must count the inputs, outputs, files and data inquiries from use-cases.
- Function points become evident using the ***use-case definition*** and activity diagram for the use-case. Each step within the activity diagram can be a transaction (inputs, outputs, files and data inquiries).



Use-Case Point /4 (traditional)

Example of Use-Case scenario:

1. The user may enter a book's ISBN number, student ID, or student name ***(input × 3)***
2. The user will press "Find" ***(input × 1)***
3. If the user enters a book ISBN number ***(input × 1)***
   - The system will display information related to that book and write the results to a file ***(output × 1) (internal file × 1)***
4. If the user entered a student name or student ID ***(input × 2)***
   - The system will return a list of all books on the waiting list for that student and write the results to a file ***(output × 1) (internal file × 1)***
   - The user can select one book from the list ***(external inquiry × 1)***
   - The system will search the database by ISBN number ***(external file × 1)***
   - The system will display information related to that book and available date and write the results to a file ***(output × 1) (internal file × 1)***

Use-Case Point /5 (another)

- Step 1: the total unadjusted actor weights (**UAW**)

Actor Type	Weighting Factor
Simple	1
Average	2
Complex	3

- Step 2: the unadjusted use case weights (**UUCW**)

Use Case Type	No of Transactions	Weighting Factor
Simple	<=3	1
Average	4 to 7	2
Complex	>=7	3

Use-Case Point /6

- Step 3: the unadjusted use case points (**UUCP**)

UUCP = UAW + UUCW

- Step 4: technical Complexity Factor (**TCF**)

TCF=0.6+(0.01*TFactor)

T1	Distributed System	T2	Response adjectives
T3	End-user efficiency	T4	Complex processing
T5	Reusable code	T6	Easy to install
T7	Easy to use	T8	Portable
T9	Easy to change	T10	Concurrent
T11	Security features	T12	Access for third parties
T13	Special training required		

Use-Case Point /7

- Step 5: the Environmental Factor (**EF**)

EF= 1.4+(-0.03*EFactor)

F1	Familiar with RUP	F2	Application experience
F3	OO experience	F4	Lead Analyst capability
F5	Motivation	F6	Stable requirements
F7	Part-time workers	F8	Difficult programming Language

- Step 6: the adjusted use case points (**UPC**)

UPC= UUCP*TCF*EF

C. Gencel et al. Functional size measurement revisited.
ACM TOSEM, 2008, 17(3)

Summary

- LOC
- Halstead software science
- Function point

Thanks for your time and attention!

练习：Halstead度量收集

为Gcc 3.4中的每个函数计算如下度量

N, V, D, E, L, T

其中, D按照第54页的估算公式计算

> 可去安装目录的子目录scripts\perl查看 c_misra_maint.pl，参考其中的函数 GetHalsteadBaseMetrics

在12月4日前提交Perl脚本以及数据集

Lexical stream

```
foreach my $lexeme ($lexer->lexemes($startline,$endline))
{
   if (($lexeme->token eq "Operator") ||
      ($lexeme->token eq "Keyword") ||
      ($lexeme->token eq "Punctuation")){
      if ($lexeme->text() !~ /[)}\]]/){
         $n1{$lexeme->text()} = 1;
         $N1++;
      }
   }
   elsif (($lexeme->token eq "Identifier") ||
          ($lexeme->token eq "Literal") ||
          ($lexeme->token eq "String")){
      $n2{$lexeme->text()} = 1;
      $N2++;
   }
}
```