

Product Models and Metrics

Product Models and Metrics

There are a large number of product types: requirements documents, specifications, design, code, specific components, test plans, ...

There are many abstractions of these products that depend on different characteristics

- logical, e.g., application domain, function

- static, e.g. size, structure

- dynamic, e.g., MTTF, test coverage

- use and context related, e.g., design method used to develop

Product models and metrics can be used to

- evaluate the process or the product

- estimate the cost of quality of the product

- monitor the stability or quality of the product over time

Product Models and Metrics

Logical Characteristics

The logical characteristic application can be measured on a nominal scale:
flight software, ground support software, ...

The logical characteristic function can be represented as

- a mathematical function abstraction for a program, e.g., $y = f(x)$, a state function abstraction for a module

or

- a nominal class, e.g., the component represents a mathematical function, data structure, ...

Product Models and Metrics

Static Characteristics

We can divide the static product characteristics into three basic classes

Size

Structure, e.g.,

Control Structure

Data Structure

Size attempts to model and measure the physical size of the product

Structure models and metrics attempt to capture some aspect of the physical structure of the product, e.g.,

Control structure metrics measure the control flow of the product

Data structure metrics measure the data interaction of the product

There are mixes of these metrics, e.g., that deal with the interaction between control and data flow.

Product Models and Metrics

Size

There are many size models and metrics, depending on the product, e.g.,
source code: lines of code, number of modules
executables: space requirements, lines of code
specification: function points
requirements: number of requirements, pages of documentation
modules: operators and operands

Size metrics can be used accurately at different points in time
lines of code is accurate after the fact but can be estimated
function points can be calculated based upon the specification

Size metrics are often used to
characterize the product
evaluate the effect of some treatment variable, such as a process
predict some other variable, such as cost

Product Models and Metrics

Lines of Code Metrics

Lines of code can be measured as:

- all source lines

- all non-blank source lines

- all non-blank, non-commentary source lines

- all semi-colons

- all executable statements

...

The definition depends on the use of the metric, e.g.,

- to estimate effort we might use all source lines as they all take effort

- to estimate functionality we might use all executable statements as they come closest to representing the amount of function in the system

Lines of code

- vary with the language being used

- are the most common, durable, cheapest metric to calculate

- are most often used to characterize the product and predict effort

Product Models and Metrics

Function Points

One model of a product is to view it as a set of interfaces, e.g., files, data passed, etc.

If a system is primarily transaction processing and the “bulk” of the system deals with transformations on files, this is a reasonable view of size.

Function Points were originally suggested as a measure of size by Al Albrecht at IBM, a means of estimating functionality, size, effort

It can be applied in the early phases of a project (requirements, preliminary design)

Product Models and Metrics

Function Points

A function point is a specific user functionality delivered by the application

It differentiates five types of files or data

- **Input type**, e.g., screen data, menu selection
- **Output Type**, e.g., report, transferred data, message
- **Query Type**, e.g., request/retrieval combination
- **File type**, e.g., database/record, indexed file
- **External interface**, e.g., reference data, external data bases

Product Models and Metrics

Function Points

There are counting rules:

Only user requested and visible components are counted

Components such as internally maintained data entries, externally maintained data entries, data maintenance activities, data output and data retrieval are categorized and valued

The final count is adjusted based upon the general characteristics of the system (distributed functions, performance considerations, complex processing)

The original function point approach was proposed by Albrecht in the late 70s in the IBM Data Processing Division

There is currently an International Function Point User Group (IFPUG) whose mission is to coordinate that the state of the practice, support users and standardize the approach

Function Point Counting Practices Manual (Version 4)

Function Points Calculation

Complexity Weights

	SIMPLE	AVERAGE	COMPLEX
Input	3	4	6
Output type	4	5	7
Query type			
— Input part	3	4	6
— Output part	4	5	7
File type	7	10	15
External interface	5	7	10

Function Points Calculation

Application Characteristics

DATA COMMUNICATIONS
DISTRIBUTED DATA OR PROCESSING
PERFORMANCE OBJECTIVES
HEAVILY-USED CONFIGURATION
TRANSACTION RATE
ON-LINE DATA ENTRY
END USER EFFICIENCY

ON-LINE UPDATE
COMPLEX PROCESSING
REUSABILITY
CONVERSION & INSTALLATION EASE
OPERATIONAL EASE
MULTIPLE-SITE
FACILITATE CHANGE



INFLUENCE SCALE

- | | |
|---|----------------------|
| 0 | NONE |
| 1 | INSIGNIFICANT, MINOR |
| 2 | MODERATE |
| 3 | AVERAGE |
| 4 | SIGNIFICANT |
| 5 | STRONG THROUGHOUT |

Function Points Calculation

FUNCTION POINTS =

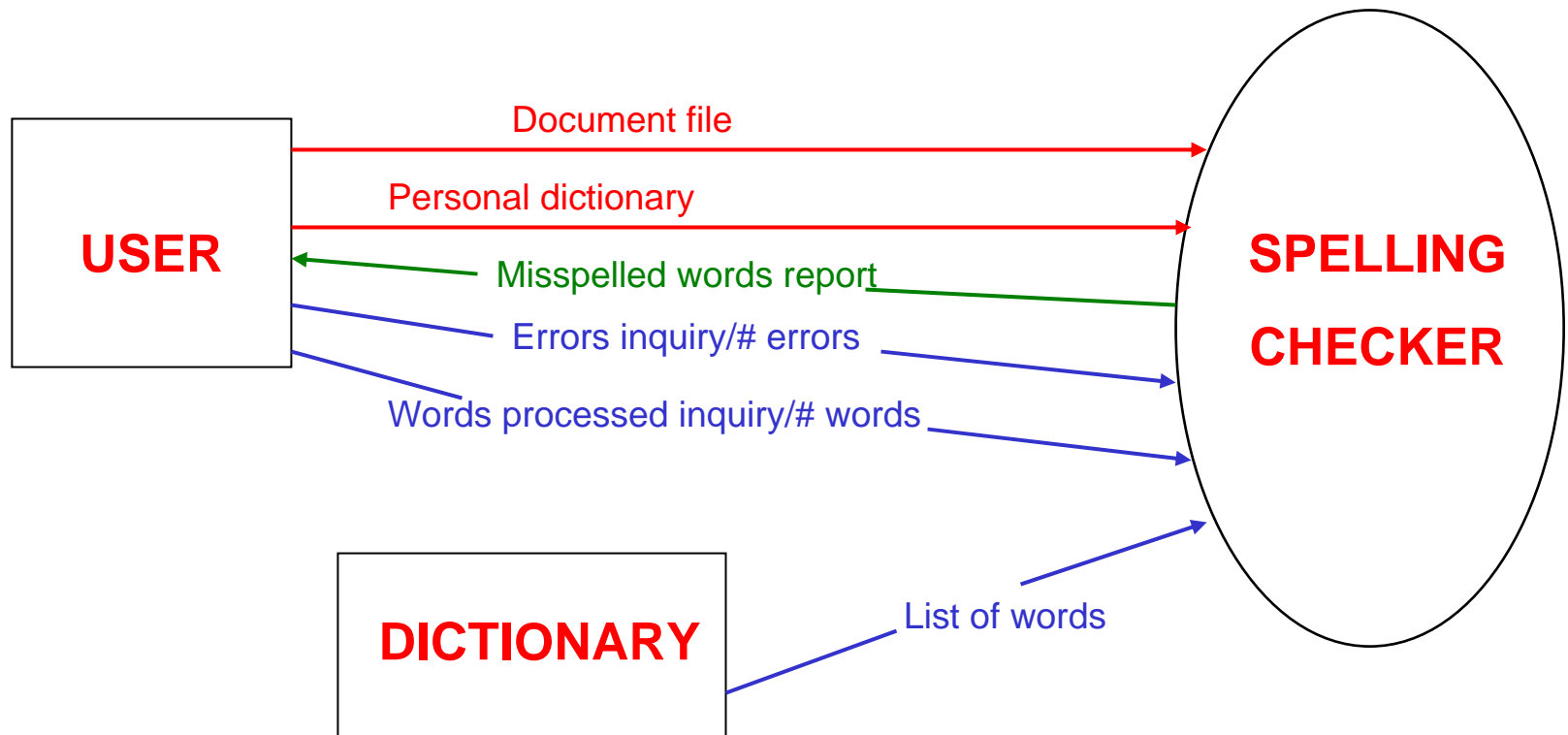
$(\sum \text{INPUTS} * \text{WEIGHTS} + \sum \text{OUTPUTS} * \text{WEIGHTS}$

$\sum \text{QUERIES} * \text{WEIGHTS} + \sum \text{FILES} * \text{WEIGHTS} +$

$\sum \text{INTERFACES} * \text{WEIGHTS}) * (0.65 + 1\% \text{ TOTAL INFLUENCE})$

Function Points Calculation Example

SPELLING CHECKER SPECIFICATION: The checker accepts as input a document file and an optional personal dictionary file. The checker lists all words not contained in either the dictionary or the personal dictionary files. The user can query the number of words processed and the number of spelling 'errors' found at any stage during the processing.



Function Points Calculation Example

- INPUTS: DOCUMENT FILE NAME, PERSONAL DICTIONARY NAME
- OUTPUT: MISSPELT WORDS REPORT, # WORDS PROCESSED
MESSAGE, # ERRORS MESSAGE
- QUERIES: ERRORS FOUND, WORDS PROCESSED
- FILES: DICTIONARY
- INTERFACES: DOCUMENT FILE, PERSONAL DICTIONARY

ASSUMING AVERAGE COMPLEXITY IN EACH CASE
AND MINOR IMPACT

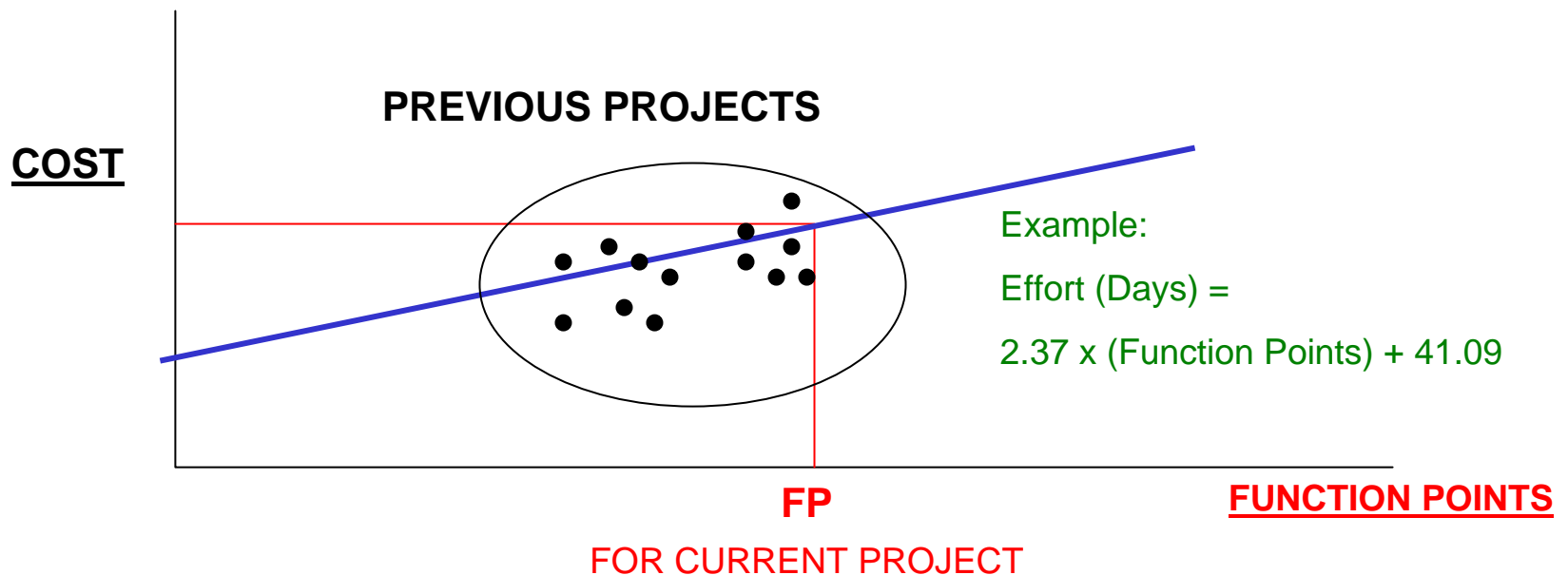
INPUTS	2	4	8
OUTPUTS	3	5	15
QUERIES	2	9	18
FILES	1	10	10
INTERFACES	2	7	14

$$65 * (0.65 + 0.01 * 14) = 51.35 \text{ FUNCTION POINTS}$$

FUNCTION POINTS

Estimating Costs

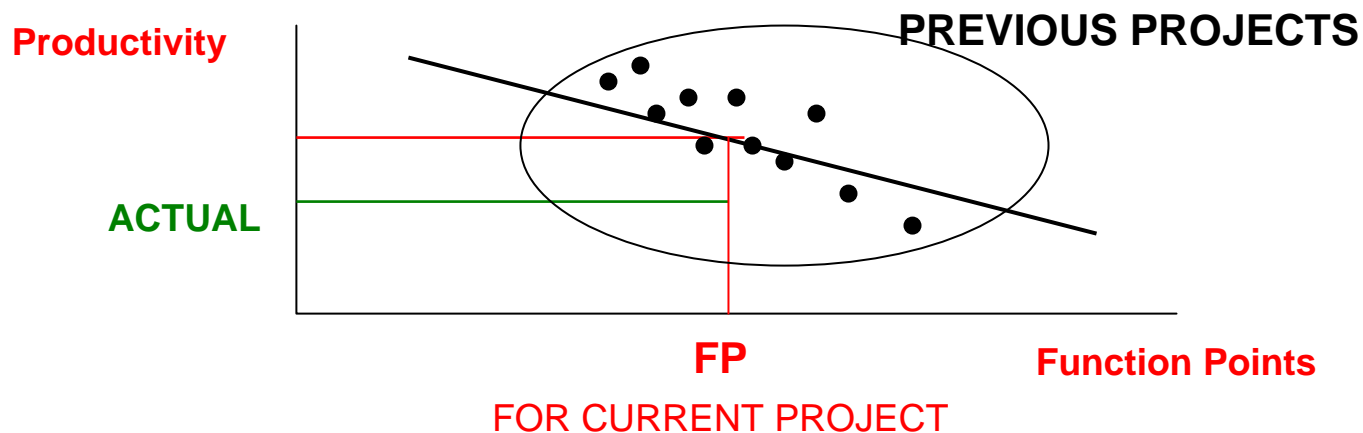
- Cost Estimation using Function Points requires
 - Cost or effort data for previous projects
 - Function Points counted in previous projects
- The estimation process
 - The cost (or effort) data of previous projects is plotted against the Function Points counted in those projects
 - This curve is used to derive of the cost of the current project from the value of its Function Points



FUNCTION POINTS

Assessing Productivity

- Productivity assessment using Function Points requires
 - productivity figures for previous projects
 - Function Points counted in previous projects
- The assessment process:
 - Data about the productivity in previous projects is plotted against the FP count in those projects
 - The expected productivity is the productivity value for the FPs of this project
 - Discrepancies between actual and expected values are analyzed



FUNCTION POINTS

Reliability Of Function Point Based Measures

- Generally a productivity model is considered good if it is capable of giving an estimate with **25% accuracy in 75% of the cases**
- Studies conducted in **MIS** (Management Information Systems) environments show that, for both development and maintenance, Function Points based measures often satisfy the criterion
- Example: Canadian financial institution study on maintenance activities (21 projects, 332 average staff days per project, min 52, max 532)

DEVIATION	PROJECTS WITHIN RANGE	
	NUMBER	%
+ / - 10%	9	43%
+ / - 20%	12	57%
+ / - 26%	17	81%

Software Science

Suppose we view a module or program as an encoding of an algorithm and seek some minimal coding of its functionality

The model would be an abstraction of the smallest number of operators and operands (variables) necessary to compute a similar function

And then the smallest number of bits necessary to encode those primitive operators and operands

This model was proposed by Maurice Halstead as a means of approximating program size.

Software Science

- MEASURABLE PROPERTIES OF ALGORITHMS

n_1 = # Unique or distinct operators in an implementation

n_2 = # Unique or distinct operands in an implementation

N_1 = # Total usage of all operators

N_2 = # Total usage of all operands

$f_{1,j}$ = # Occurrences of the j^{th} most frequent operator
 $j = 1, 2, \dots, n_1$

$f_{2,j}$ = # Occurrences of the j^{th} most frequent operand
 $j = 1, 2, \dots, n_2$

THE VOCABULARY n IS

$$n = n_1 + n_2$$

THE IMPLEMENTATION LENGTH IS

$$N = N_1 + N_2$$

and

$$N_1 = \sum_{j=1}^{n_1} f_{1,j}$$

$$N_2 = \sum_{j=1}^{n_2} f_{2,j}$$

$$N = \sum_{i=1}^2 \sum_{j=1}^{n_i} f_{ij}$$

Example: Euclid's Algorithm

LAST:

```
IF (A = 0)
BEGIN GCD := B; RETURN END;
IF (B = 0)
BEGIN GCD := A; RETURN END;
```

HERE:

```
G := A/B; R := A - B X G;
IF (R = 0) GO TO LAST;
A := B; B := R; GO TO HERE
```

Operator Parameters

Greatest Common Divisor Algorithm

OPERATOR	j	f_{1j}
;	1	9
:=	2	6
() or BEGIN...END	3	5
IF	4	3
=	5	3
/	6	1
-	7	1
x	8	1
GO TO HERE	9	1
GO TO LAST	10	1
	$n_1 = 10$	$N_1 = 31$

Operand Parameters

Greatest Common Divisor Algorithm

OPERAND	j	f_{2j}
B	1	6
A	2	5
O	3	3
R	4	3
G	5	2
GCD	6	2
	$n_2 = 6$	$N_2 = 21$

Software Science Metrics

PROGRAM LENGTH:

$$N \sim \hat{N} = n_1 \log_2 n_1 + n_2 \log_2 n_2$$

\hat{N} = The number of bits necessary to represent all things that exist in the program at least once

\hat{N} = The number of bits necessary to represent the symbol table

PROGRAM VOLUME: (Size of an implementation)

$$V = N \log_2 n$$

B = The number of bits necessary to represent the program

POTENTIAL VOLUME: (Minimal size of an implementation)

$$V^* = (2 + n^*_2) \log_2 (2 + n^*_2)$$

Where n^*_2 represents the number of input/output parameters

V^* = A measure of the specification for an algorithm

Software Science Metrics

PROGRAM LEVEL: (Level of an implementation)

$$L = V^* / V$$

$$D = 1/L = \text{Difficulty}$$

PROGRAMMING EFFORT:

$$E = V D = V/L = V^2/V$$

E = The effort required to comprehend an implementation rather than produce it

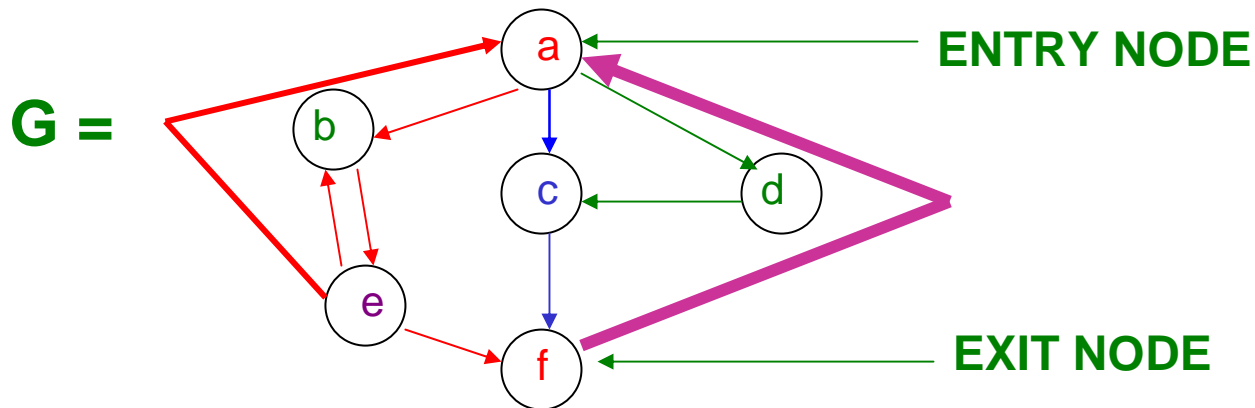
E = A measure of program clarity

CYCLOMATIC COMPLEXITY

- The **Cyclomatic Number** $V(G)$ of a graph G with n vertices, e edges, and p connected components is

$$v(G) = e - n + p(2)$$

- In a strongly connected graph G , the cyclomatic number is equal to the maximum number of linearly independent circuits



- $V(G) = 9 - 6 + 2 = 5$ linearly independent circuits, e.g.,
(a b e f a), (b e b), (a b e a), (a c f a), (a d c f a)

CYCLOMATIC COMPLEXITY

Suppose we view a program as a directed graph, an abstraction of its flow of control, and then measure the complexity by computing the number of linearly independent paths, $v(G)$

Properties of Cyclomatic Complexity

- 1) $v(G) \geq 1$
- 2) $v(G) = \#$ linearly independent paths in G ; it is the size of a basis set
- 3) Inserting or deleting functional statements to G does not affect $v(G)$
- 4) G has only one path iff $v(G) = 1$
- 5) Inserting a new edge in G increases $v(G)$ by 1
- 6) $v(G)$ depends only on the decision structure of G

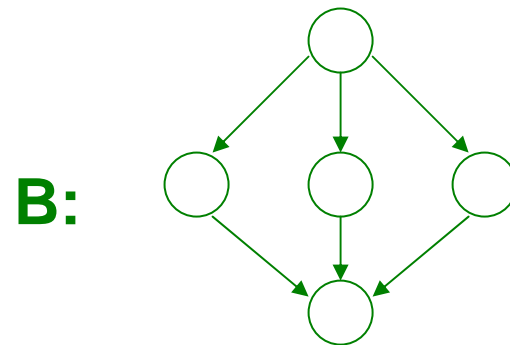
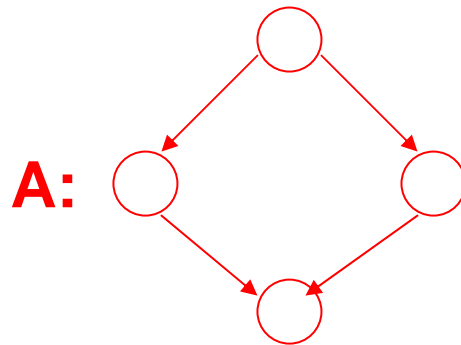
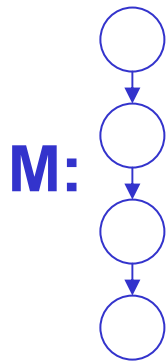
CYCLOMATIC COMPLEXITY

- For a collection of components
 - The cyclomatic number of the collection is the sum of the cyclomatic numbers of the individual components

$$v(C) = \sum_i v(C_i), \text{ where } C = \cup C_i$$

- In the example: For more than 1 component

$$v(M \cup A \cup B) = e - n + 2p = 13 - 13 + 2(3) = 6$$



CYCLOMATIC COMPLEXITY

It can be shown that

$$V(G) = \text{NUMBER OF DECISIONS} + 1$$

The concept of cyclomatic complexity
tied to to complexity of testing program
easy to compute
has been well studied

Cyclomatic complexity is used mostly on modules

It has been recommended that the cyclomatic complexity of a module
be kept small, e.g. less than 10

There is some evidence to support this

SEL

Evaluating and Comparing Software Metrics

GOALS

Do measures like Cyclomatic Complexity and the software science metrics relate to effort and quality?

Does the correspondence increase with greater accuracy?

How do these metrics compare with traditional size metrics such as the number of source lines of code or the number of executable statements?

How do these metrics relate to one another?

DEFINITIONS

EFFORT: The number of staff hours programmers and managers spend from the beginning of functional design to the end of acceptance testing.

QUALITY: The number of program faults reported during the development of the product.

Metric Evaluation in the SEL

Size and Complexity Measures Investigated

The Data:

Commercial software: Satellite Ground support software

Systems consist of 51K to 112K lines of FORTRAN source code

Ten to sixty one percent of the source code was modified from previous projects

Development effort ranges from 7K to 22K staff hours

This analysis focuses on:

Data from 7 projects

only newly developed modules (i.e., subroutines, functions, main procedures and block data)

Metric Evaluation in the SEL Size and Complexity Measures Investigated

Objective Size And Complexity Measures Investigated

Source Lines Of Code

Source Lines Of Code Excluding Comments

Executable Statements

Software Science Metrics

N : Length In Operators And Operands

V : Volume

V* : Potential Volume

L : Program Level

E : Effort

B : Bugs

Cyclomatic Complexity

Cyclomatic Complexity Excluding Compound Decisions

(Referred To As Cyclo_cmplx_2)

Number Of Procedure And Function Calls

Calls Plus Jumps

Revisions (Versions) Of The Source In The Program Library

Number Of Changes To The Source Code

Metric Evaluation in the SEL

Size and Complexity Measures Investigated

Spearman Correlations (R Values - All Signif. At P = 0.001)

	All Projects	Single Project		Single Programmer	
Validity Ratio	All	All	80%	90%	92.5%
# Modules	731	79	29	20	31
E^^	.49	.70	.75	.80	.79
CYCLO_CMLPX_2	.47	.76	.79	.79	.68
CALLS & JUMPS	.49	.78	.81	.82	.70
SOURCE LINES	.52	.69	.67	.73	.86
EXECUT. STMTS	.46	.69	.71	.78	.75
V	.45	.68	.72	.80	.68
REVISIONS	.53	.68	.72	.80	.68

Some Relation To Effort Across All Projects

Relation Improve With:

Individual Projects

Validated Data

Individual Programmers

Metric Evaluation in the SEL

Size and Complexity Measures Investigated

The Number Of Program Faults For A Given Module Is The Number Of System Changes That Listed The Module As Affected By An Error Correction

Weighted faults (w_flts) is A measure Of The amount Of effort spent isolating And fixing faults In A module

Spearman Correlation (R Values - All Signif. At P = 0.0001, Except (*) Signif. At P = 0.05)

ALL	SINGLE PROJECTS		SINGLE PROJECT		PROGRAMMER	
MODULES	652		132		21	
	FAULTS	W_FLTS	FAULTS	W_FLTS	FAULTS	W_FLTS
E^^	.16	.19	.58	.52	.67	.65
CYCLO_CMPLX_2	.19	.20	.55	.49	.48*	.45*
CALLS & JUMPS	.24	.25	.57	.52	.60*	.56*
SOURCE LINES	.26	.27	.65	.62	.66	.65
EXECUT. STMTS	.18	.20	.54	.51	.58*	.53*
B	.17	.19	.54	.50	.68	.66
REVISIONS	.38	.38	.78	.69	.83	.81
EFFORT	.32	.33	.64	.62	.67	.62

Relations Low Overall; # Revisions Strongest
Relations Improve With Individual Projects Or Programmers

Metric Evaluation in the SEL Size and Complexity Measures Investigated

Conclusions

Used commercially obtained data to validate software metrics

Common environment

Was able to perform validity checks and accuracy ratings

No metric

seemed to satisfactorily explain effort or development faults

related convincingly better with effort than the others

The strongest effort correlations

come from individual programmers or certain validated projects

increase with the more reliable data

The number of revisions correlates with development faults better than any other metric

Many size and complexity measures relate well with each other

Product Models and Metrics

Data Structure Metrics

Data structure metrics measure the data interaction of the product

Major Concepts:

Coupling: refers to the degree of **interdependence between** parts of a design, usually the amount of data shared by parts

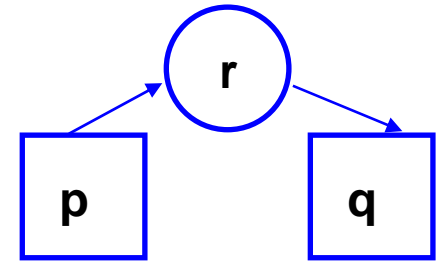
Cohesion: refers to the degree of **dependence within** parts of the design, usually the strength of the interaction

The definitions of these concepts depends on the design paradigm or notation used to expressed the design. It is often programming language dependent

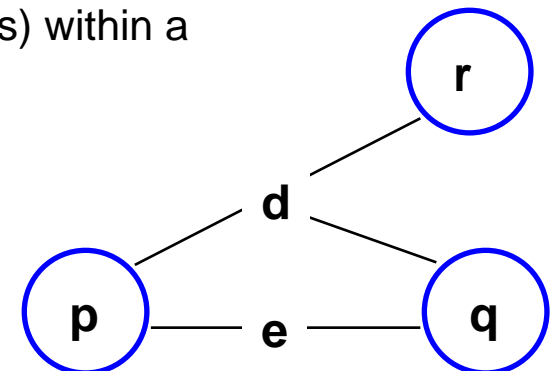
Data Structure Metrics: Data bindings

- A SEGMENT - GLOBAL - SEGMENT DATA BINDING (p, r, q) is an occurrence of the following:

- (1) segment p modifies global variable r
- (2) variable r is accessed by segment q
- (3) $p \neq q$



- Existence of a data binding (p, r, q) \Rightarrow q dependent on the performance of p because of r
- $DB (p, r, q) \neq DB (q, r, p)$
- (p, r, q) represents a unique communication path between p and q
- The total # Data Bindings represents the degree of a certain kind “connectivity” (i. e. , Between segment pairs via globals) within a complete program



Data Structure Metrics: Data bindings

INT A, B, C, D

```
PROC P1
  /* USES A, B */
  . . . . .
PROC P2
  /* USES A, B */
  . . . . .
  CALL P3 (X)
  . . . . .
```

DATA BINDINGS

(P1, A, P2)

(P1, B, P2)

(P3, C, P4)

(P3, D, P4)

(P2, E, P3)

```
PROC P3 (INT E)
  /* USES C, D */
  . . . . .
PROC P4
  /* USES C, D */
  . . . . .
```

```
[ [ P1 P2
  [ P3 P4
```

Data Structure Metrics: Data bindings

Levels of data binding (DB)

- **Potential DB** is an ordered triple (p, x, q) for components p and q and variable x in the scope of p and q
- **Usage DB** is a potential DB such that p and q both use x for reference or assignment
- **Feasible DB** (actual) is a usage DB such that p assigns to x and q references x
- **Control flow DB** is a feasible DB such that flow analysis allows the possibility of q being executed after p has started

Data Structure Metrics

Segment Global Usage Pairs

- A segment-global usage pair (p, r) is an instance of a global variable r being used by a segment p (i.e., r is either modified or set by p).
- Each usage pair represents a unique “**use connection**” between a global and a segment
- Let actual usage pair **(AUP)** represent the count of true usage pairs (i.e., r is actually used by p)
- Let possible usage pair **(PUP)** represent the count of potential usage pairs (i.e., given the program’s globals and their scopes, the scope of r contains p so that p could potentially modify r) (worst case)
- Then the relative percentage usage pairs **(RUP)** is $RUP = AUP/PUP$ and is a way of normalizing the number of usage pairs relative to the problem structure
- The **RUP** metric is an empirical estimate of the likelihood that an arbitrary segment uses an arbitrary global

Measurement Across Time

Measures are sometimes difficult to understand in the absolute

However the relative changes in metrics over the the evolution of the system can be very informative

This evolution may be within one development cycle of the product

e.g., **Requirements** → **Design** → **Code** → ...

Or

Multiple versions of the same product

e.g., **Code₁** → **Code₂** → **Code₃** → ...

Measurement Across Time Development/Maintenance Vector

A **vector of metrics**, m_1, m_2, m_n can be defined dealing with various aspects of the product, i.e., effort, changes, defects, logical, physical, and dynamic attributes, environmental considerations, ...

For example, some physical attributes might include

(decisions, interaction of data, interaction of data, size)
across modules within a module

The vector characterizes the product at some point in time

We can view it at various stages of product evolution to monitor how the product is changing

We can provide a set of bounds for the metrics to signal potential problems and anomalies

Measurement Across Time Case Study

Various metrics were used during different points in the development of a software product

The product was a compiler for a structured programming language

- about 6,500 high level source statements
- about 17,000 lines of source code

We will examine the changes of the values of various metrics across time

- to provide insight into how the product was progressing
- to allow us to evaluate the quality of the product

There were 17 enhancements of the product for this study

- we will look at 5 major iterations
- there were iterations after the last one

Measurement Across Time Case Study

Statistics from Compilers at 5 Selected Points in the Iterative Process

	<u>1</u>	<u>2</u>	<u>3</u>	<u>4</u>	<u>5</u>
• Number of statements	3404	4217	5181	5847	6350
• Number of procedures and functions	89	189	213	240	289
• Number of separate Compiled modules	4	4	7	15	37
• Average nesting level	3.4	2.9	2.9	2.9	2.8
• Average number of Tokens per statement	5.7	6.3	6.6	7.2	7.3

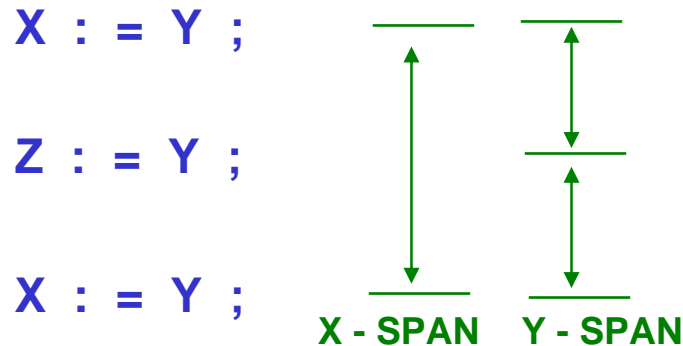
Measurement Across Time Case Study

Statistics from Compilers at 5 Selected Points in the Iterative Process

	<u>1</u>	<u>2</u>	<u>3</u>	<u>4</u>	<u>5</u>
Usage count of (Segment, Global) pairs (AUP)	611	786	941	1030	974
Total possible count of (Segment, Global) pairs	4128	8707	10975	6930	4584
Percentage use of globals (PUP)	14.8	9.0	8.6	14.9	21.2
Total token size	19403	26567	34194	42098	46355
Number of Data Bindings	2610	6662	8759	12006	10442
Number of Data Bindings per Thousand tokens	13.4	17.7	25.6	28.5	22.5

Psychological Complexity: SPAN

- A **SPAN** is the number of statements between two textual references to the same identifier



- **SPAN (X)** = count of # statements between first and last statements (assuming no intervening references to X) Y has two spans
- For n appearances of an identifier in the source text , n - 1 spans are measured
- All appearances are counted except those in declare statements
- If **SPAN > 100** statements, then one new item of information must be remembered for 100 statements till read again

Psychological Complexity: SPAN

- COMPLEXITY ~ # SPANS at any point (take max, average, median)

OR ~ # Statements a variable must be remember
(on the average) [average span]

VARIATION

- Do a live/dead variable analysis
- Complexity proportional to # variables alive at any statement
- How does one scale up this measure?

$$C(M) = \frac{\# \text{ STMTS}}{\sum_{j=i} \frac{n_i \cdot s(n_i)}{\# \text{ stmts}}}$$

Where $n_i = \#$ spans of size $S(n_i)$

Psychological Complexity: Variable SPAN

Variable span has been shown to be a reasonable measure of complexity

For commercial PL/1 programs, one study showed that a programmer must remember approximately 16 items of information when reading a program

Product Models and Metrics

Dynamic Characteristics

We can divide the dynamic product the characteristics into two basic classes

We can view them as checking on the
Behavior of the input to the code, e.g., coverage metrics
Behavior of the code itself, e.g., reliability metrics

PRODUCT METRICS

Coverage Metrics

Based upon checking what aspects of the product are effected by a set of inputs

For example,

procedure coverage - which procedures are covered by the set of inputs

statement coverage - which statements are covered by the set of inputs

branch coverage - which parts of a decision node are covered by the set of inputs

path coverage - which paths are covered by the set of inputs

requirements section coverage - which parts of the requirements document have been read

Used to

check the quality of a test suite

support the generation of new test cases

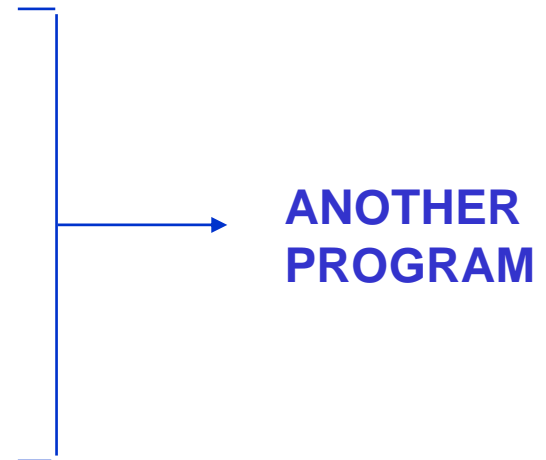
Test Coverage Metrics

PASCAL

# TEST CASES:	<u>32</u>	<u>36</u>
SUBPROGRAM COVERAGE	.81	.92
BRANCH PATH COVERAGE	.59	.67
I/O COVERAGE	.23	.54
DO LOOP ENTRY	.92	.94
ASSIGNMENT	.85	.91
OTHER EXECUTABLE	.74	.78
CODE COVERAGE	.70	.80

FORTRAN

# TEST CASES:	<u>68</u>
SUBROUTINE COVERAGE	.91
FUNCTION COVERAGE	1.00
BRANCH PATH	.63
I/O	.35
DO-LOOP	.74
ASSIGNMENT	.48
OTHER EXECUTABLE	.66



Requirements Coverage

Enumerate the requirements

(e.g., worst case: assign a number to every sentence in the Requirements Document)

Let the requirements be $R_1, R_2, R_3, \dots R_n$

Enumerate the modules in the system, $M_1, M_2, M_3, \dots M_p$

Then we can build a **Requirement X Module Matrix**, where an x in (R_i, M_j) implies that the requirement is implemented in full or in part by that module

Coverage of all the modules that implement a requirement, $R_i \rightarrow$ coverage of R_i

Requirements Coverage

We can be more sophisticated

let (R_i, M_j) be the percent of R_i covered by M_j

Note that the sum of the row is = 100

let (R_i, M_j) be the percent M_j covers R_j

Note that the sum of the row is ≥ 100

	Module	
Requirement	percent R_i covered by X_j	
	percent X_i covers R_j	

Consider other coverage matrices, e.g.,

Module x Test Case Matrix where an element is marked with an x if the test covers the module

Then **$R \times M$** multiplied by **$M \times T$** tells you which tests cover which requirements

Customer Coverage

We can also consider customer coverage, i.e., it is assumed that different customers use different requirements with different probabilities.

We can calculate a **Requirements X Customer Usage Matrix**, e.g.,

		CUSTOMER	
REQUIREMENTS		percent use	
		by customer	

Then we can calculate the Customer x Test Case matrix by multiplying the Customer Usage x Requirements with the Requirements x Module with the Module x Test to determine what test are most representative of which customers

Coverage Matrices

Coverage matrices provide a kind of traceability from the customer to the set of test cases.

The matrices are useful feedback during development.

e.g., R x M provides some indication of the independence of various requirements with respect to modules that implement them

RELIABILITY

How Can We Use Reliability Metrics

System engineering

Determine the best trade-off between reliability and cost, schedule, etc.

Optimize life cycle cost

Specify reliability to the designer

Project management

Progress monitoring

Scheduling

Investigation of alternatives

Operational software management

Evaluation of software engineering management

Musa, Goel, Littlewood

RELIABILITY

Software Reliability Models

Time dependent approaches

Time between failures (Musa model)

Failure counts in specified intervals (Goel/Okumoto)

Time-independent approaches

Error seeding

Input domain analysis

Problems with use of reliability models

Lack of clear understanding of inherent strengths

And weaknesses

Underlying assumptions and outputs not fully

Understood by user

Not all models applicable to all testing environments

RELIABILITY MODELS

Musa

Assumptions:

1. Errors are distributed randomly through the program
2. Testing is done with repeated random selection from the entire range of input data
3. The error discovery rate is proportional to the number of errors in the program
4. All failures are traced to the errors causing them and corrected before testing resumes
5. No new errors are introduced during debugging

$$T = \frac{1}{KE} e^{Kt}$$

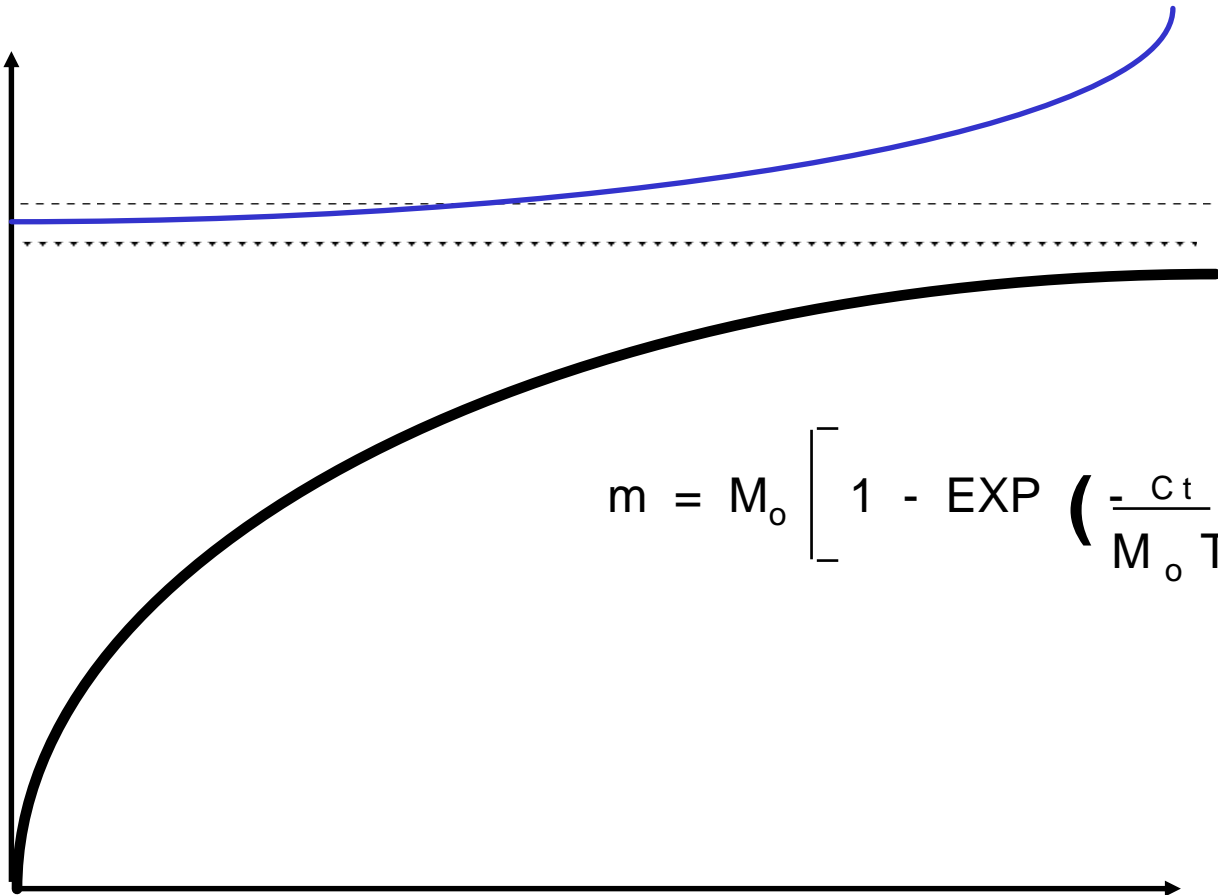
WHERE **E** is total errors in the system
t is the accumulated run time (starts @ 0)
T is the mean time to failure

RELIABILITY

Failures Experienced vs Cumulative Execution Time

FAILURES
EXPERIENCED
 m

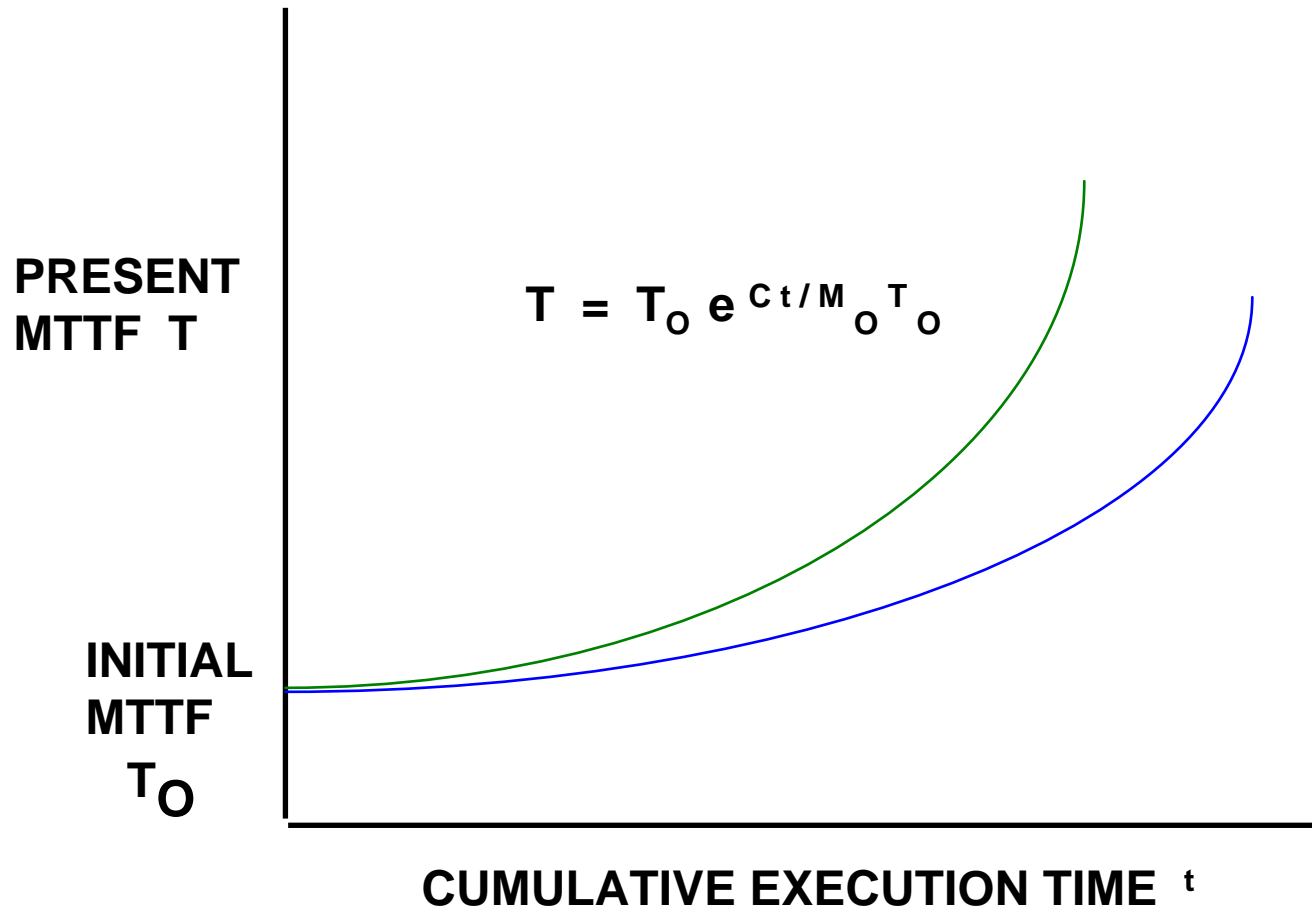
TOTAL
FAILURES
 M_o



CUMULATIVE EXECUTION TIME t

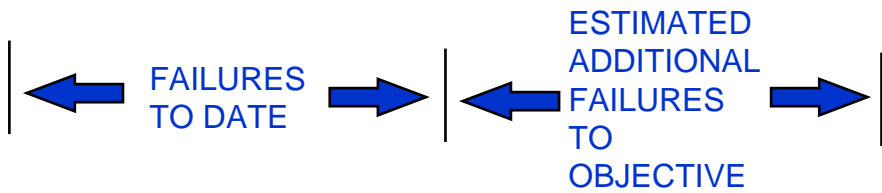
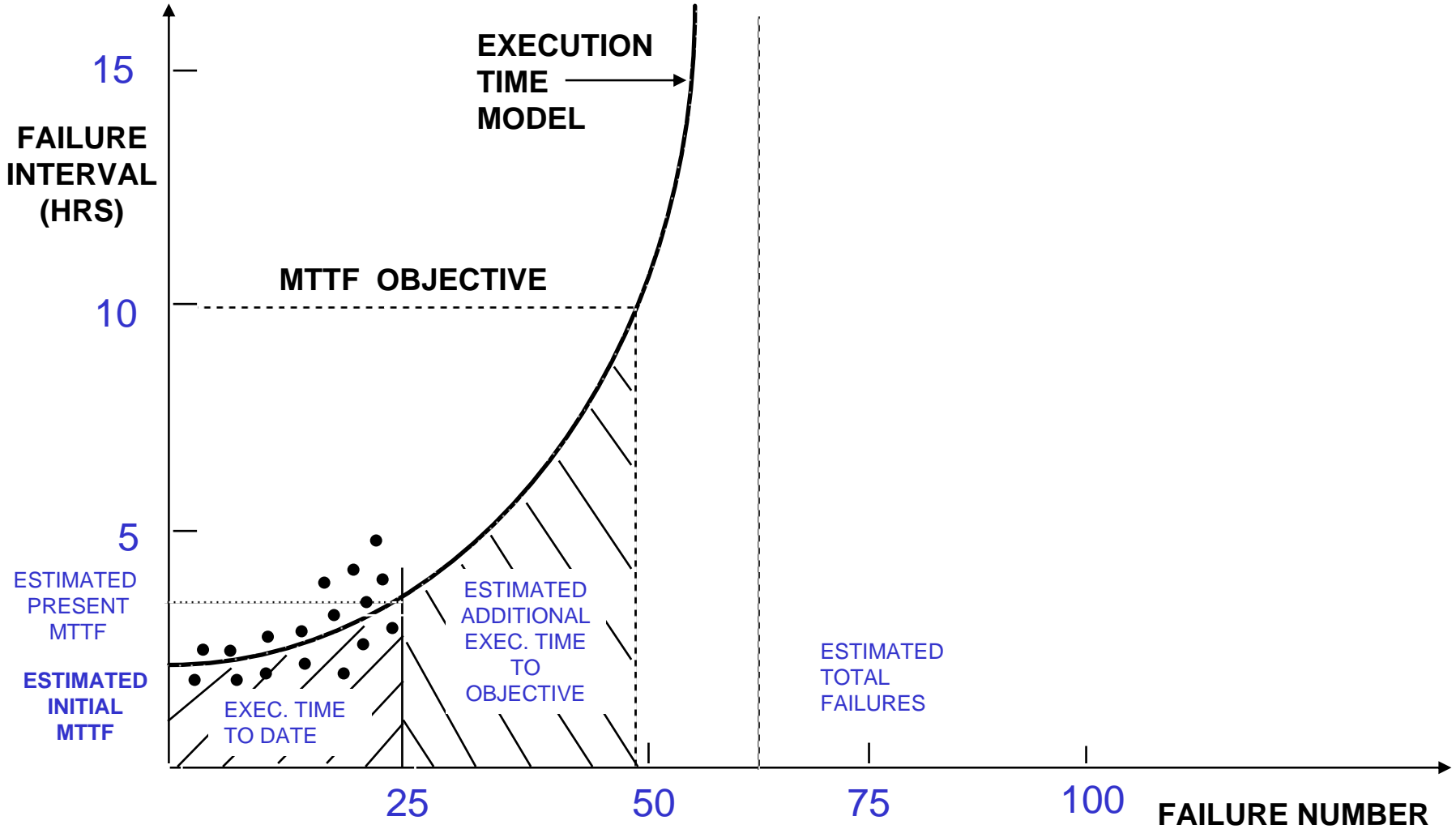
RELIABILITY

Present MTTF vs Cumulative Execution Time



RELIABILITY

Software Reliability Estimation Execution Time Model



RELIABILITY

Combination of Approaches

Clean room

Developer uses reading techniques, top down development

Testing done by independent organization at incremental steps

Reliability model used to provide developer with quality assessment

Functional testing/coverage metrics

Use functional testing approach

Collect error distributions, e.g., Omission vs commission

Obtain coverage metrics

Knowing number of errors of omission, extrapolate

Error analysis and reliability models

Establish error history from previous projects

Distinguish similarities and differences to current project

Determine prior error distributions for the current project

Select a class of stochastic models for the current project

Update prior distributions and compare actual data with the priors for the current project

Acceptance Test Plan Evaluation

Goal: How good is the acceptance test plan? Does it represent actual use?

Environment: NASA SEL, Subset of a large satellite system

Product dimensions:

What is the size of the system?

(68 Fortran subroutines; 10,000 lines of code; 4,300 executable statements)

What is the size of the test suite?

(10 multi-part acceptance tests,
not a rigorous sampling of input domain but not trivial)

What are the number of operational uses? (60 uses)

Changes/defects:

How many faults were found during acceptance test?

How many faults were found during operational use? (8)

Acceptance Test Plan Evaluation

Quality perspective: Compare the structural coverage of the acceptance tests and operational use of the system.

What is the procedure coverage and statement coverage for the acceptance test suite by test and in total?

What is the % of unique code exercised by each test?

What is the procedure coverage and statement coverage for the operational use of the system?

What is the overlap of the acceptance test and operational use coverage?

Is there anything different about the statements executed in operational test but not covered during acceptance test?

Acceptance Test Plan Evaluation

Structural Coverage of Acceptance Test

Executable Statement Coverage by 10 Test Cases

Case	Procedures Executed (%)	Executable Statements (%)	% Unique Code
t1	50.0	27.5	0.0
t2	50.0	27.2	0.0
t3	48.5	24.4	0.0
t4	60.3	37.9	4.4
t5	69.1	47.1	1.7
t6	67.6	42.7	0.0
t7	66.2	39.0	0.0
t8	66.2	45.6	1.0
t9	66.2	41.0	0.0
t10	66.2	40.2	0.0
Cumulative	75.0	56.0	
Intersect	42.6	18.1	

44% of executable statements were not exercised in acceptance test.
They may have been executed in system/unit testing

Acceptance Test Plan Evaluation

Structural Coverage of Operational Use

Structural Coverage of 60 Operational Usage Cases

	Procedures Executed (%)	Executed Statements (%)
Cumulative	80.0	64.9
Intersection	27.9	10.3

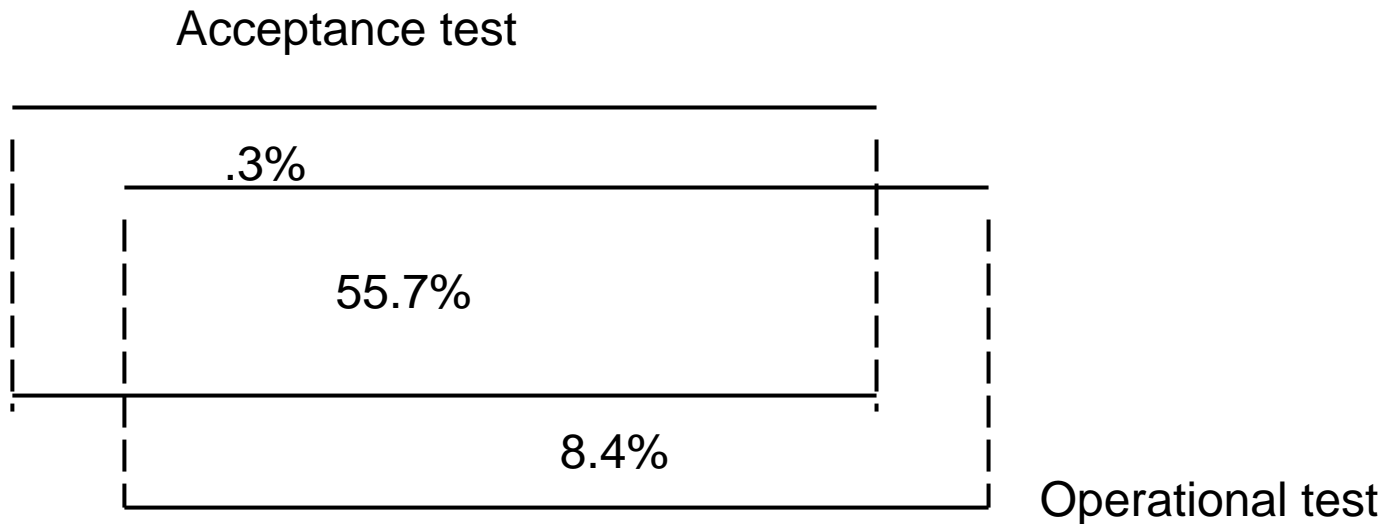
10% of the code was executed by all of the operational cases

Acceptance Test Plan Evaluation

Are Acceptance Tests representative of operational usage?

Must be true if acceptance test failures used to predict operational failures

Coverage:



Representation:

The mix of statements in the 8.4% and 55.7% differ

Twice as likely to execute a call or if in the 8.4%

Otherwise can't distinguish by structural coverage numbers

However, no faults were revealed in the 8.4%

Acceptance Test Plan Evaluation

Observations

Acceptance test plan reasonably effective, but could be refined for future releases.

About 56% of code exercised by acceptance tests; 65% by operational use.

Acceptance test reasonably representative of operational tests, no faults found in unexercised code.

If acceptance tests randomized, reliability models may be used to predict operational reliability with moderate chance of success.

Can these results about the test plan be generalized to evaluate the test plan process?

Automatable Change And Error Metrics

Automatable Metric

No interference to the developer

Computed algorithmically from quantifiable sources

Reproducible on other projects with the same algorithms

Useful Metric

Sensitive to externally observable differences in the development environment

Relative values correspond to some intuitive notion about characteristic differences in the environment

Examples

Program changes: Textual revision in the source code representing one conceptual change

Job steps: The number of computer accesses during development or maintenance

Automatable Change And Error Metrics

Program Changes

Textual revisions in the source code of a module during the development period
One program change should represent one conceptual change to the program

A program change is defined as:

- One or more changes to a single statement
- One or more statements inserted between existing statements
- A change to a single statement followed by the insertion of new statements

The following are not counted as program changes:

- The deletion of one or more existing statements
- The insertion of standard output statements or special compiler-provided debugging directives
- The insertion of blank lines or comments, the revision of comments and reformatting without alteration of existing statements

Program changes have been shown to correlate well with faults

Automatable Change And Error Metrics

Job Steps

The number of computer accesses

A single programmer-oriented activity performed on the computer at the operating system command level

Basic to the development effort and involves non-trivial expenditures of computer or human resources

Examples: text editing, module compilation, program compilation, link editing, program execution

CONTROLLED EXPERIMENT

GOAL: Evaluate the effect of a disciplined approach to software development

HYPOTHESES: The disciplined approach reduces the average cost and complexity of the process

The disciplined team should behave more like an individual programmer than a team in terms of the resulting product (conceptual integrity)

DISCIPLINED APPROACH: Top down design, process design language, walk-throughs, chief programmer teams, librarian

SUBJECTS:

7	three-person disciplined teams (dt)
6	three-person ad hoc teams (at)
6	ad hoc individuals (ai)

PROJECT: Compiler project
1200 source lines of code

PROGRAM_CHANGES

<u>ACTUAL DATA (ORDERED):</u>	<u>AVERAGE VALUES:</u>	<u>RESULTS:</u>
$DT_4 = 111$	$DT = 159$	$DT < AT = AI$
$DT_7 = 114$	$AI = 353$	
$DT_2 = 120$	$AT = 522$	
$DT_3 = 136$		
$DT_6 = 159$		
$AI_6 = 187$		
$DT_1 = 223$		
$DT_5 = 251$		
$AI_3 = 270$		
$AI_2 = 281$		
$AT_6 = 287$		
$AT_1 = 301$		
$AI_4 = 316$		
$AT_4 = 394$		
$AT_5 = 493$		
$AI_5 = 525$		
$AI_1 = 539$		
$AT_2 = 554$		
$AT_2 = 1107$		

JOB STEPS

ACTUAL ORDERING WAS:

$$DT_2 = 44$$

$$DT_6 = 58$$

$$DT_1 = 67$$

$$DT_3 = 68$$

$$DT_4 = 79$$

$$AI_6 = 87$$

$$DT_3 = 90$$

$$DT_7 = 123$$

$$AT_5 = 150$$

$$AI_3 = 151$$

$$AI_1 = 159$$

$$AT_6 = 164$$

$$AT_4 = 173$$

$$AI_5 = 176$$

$$AI_4 = 183$$

$$AT_1 = 216$$

$$AT_3 = 266$$

$$AI_2 = 351$$

$$AT_2 = 372$$

CATEGORIES:

TOTAL JOB STEPS, MODULE
COMPILATIONS, PROGRAM
EXECUTIONS, ETC.

AVERAGE NUMBER:

$$DT = 76$$

$$AI = 186$$

$$AT = 224$$

RESULTS:

$$DT < AT = AI$$

CYCLOMATIC COMPLEXITY

FOUR VARIATIONS:

SIMPPRED-NCASE -- SIMPLE PREDICATES CONTRIBUTES 1 UNIT;
CASE STATEMENTS CONTRIBUTE 1 UNIT FOR EACH CASE LABEL.

SIMPPRED-LOGCASE -- SIMPLE PREDICATES CONTRIBUTE 1 UNIT;
CASE STATEMENTS CONTRIBUTE $\lfloor \log_2(n) \rfloor$ UNITS, WHERE n IS THE NUMBER OF CASE LABELS.

COMPPRED-NCASE -- COMPOUND PREDICATES CONTRIBUTE 1 UNIT;
CASE STATEMENTS CONTRIBUTE 1 UNIT FOR EACH CASE BRANCH; MULTIPLE CASE LABELS ON THE SAME CASE BRANCH ARE DISREGARDED.

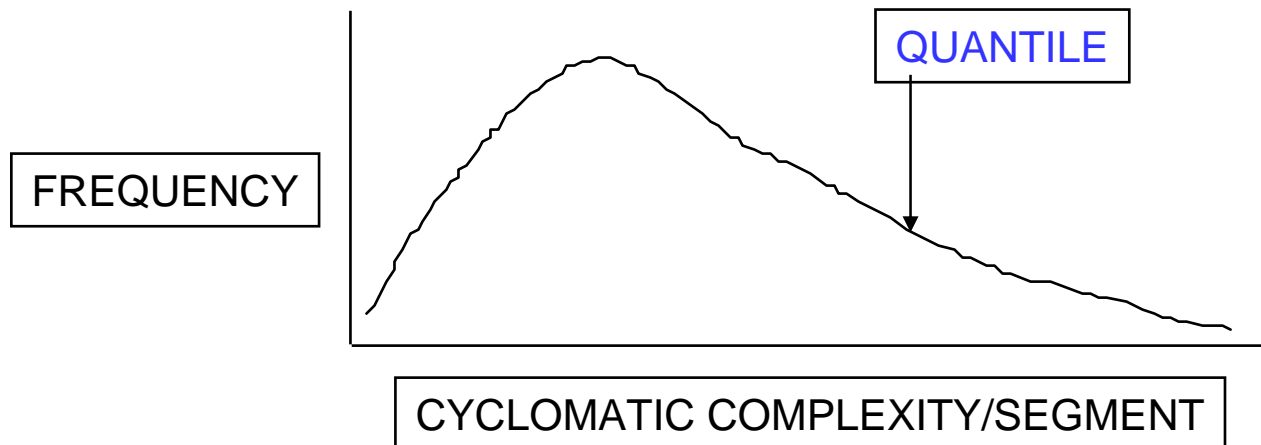
COMPPRED-LOGCASE -- COMPOUND PREDICATES CONTRIBUTE 1 UNIT;
CASE STATEMENTS CONTRIBUTE $\lfloor \log_2(n) \rfloor$ UNITS, WHERE n IS THE NUMBER OF CASE BRANCHES; MULTIPLE CASE LABELS ON THE SAME CASE BRANCH ARE DISREGARDED.

CYCLOMATIC COMPLEXITY

SUM OF CYCLOMATIC COMPLEXITY ACROSS SEGMENTS SIMILAR TO DECISIONS RESULT

NUMBER OF SEGMENTS EXCEEDING CYCLOMATIC COMPLEXITY OF 10 NO SIGNIFICANT RESULTS

DISTRIBUTION ACROSS PRODUCT OF CYCLOMATIC COMPLEXITY PER SEGMENT



STUDY RESULTS

Process measures:

Job steps (effort):

For all categories of job step

$$DT < AT = AI$$

Program changes (errors):

$$Dt < at = ai$$

PRODUCT MEASURES:

SIZE:	SEGMENTS	$AI < DT = AT$
	LINES	$AI < DT < AT$
	DECISIONS	$AI = DT < AT$

COMPLEXITY:

Comparing cyclomatic complexity of modules in Upper QUANTILES

When case statement contributes n

$$DT = AT < AI$$

When case statement contributes $\log n$

$$DT < AT = AI$$

CONCLUSIONS:

Hypotheses support

Can quantitatively demonstrate the effectiveness of the approach

Product Models and Metrics

References

- Victor R. Basili and Albert. J. Turner, Iterative Enhancement: A Proactical Technique for Software Development, IEEE Transactions on Software Engineering, vol. 1, #4, December 1975.
- Victor R. Basili and Robert Reiter, Jr., An Investigation of Human Factors in Software Development, IEEE Computer Magazine, pp 21-38, December 1979.
- Victor R. Basili & Robert Reiter, Jr., A Controlled Experiment Quantitatively Comparing Software Development Approaches, IEEE Transactions on Software Engineering, pp 299-320, May 1981.
- Victor R. Basili and David Hutchens, An Empirical Study of a Syntactic Complexity Family, IEEE Transactions on Software Eng. , vol. SE-9, #6, pp 664-672, November 1983.
- Victor R. Basili, Richard Selby and Tsai-Yun Phillips, Metric Analysis and Data Validation Across FORTRAN Projects, IEEE Transactions on Software Engineering, vol. SE-9, #6, pp 652-663, November 1983.
- David H. Hutchens and Victor R. Basili, System Structure Analysis: Clustering with Data Bindings, IEEE Transactions on Software Engineering, pp 749-757, August 1985.
- Maurice Halstead, Elements of Software Science, North Holland, 1979.

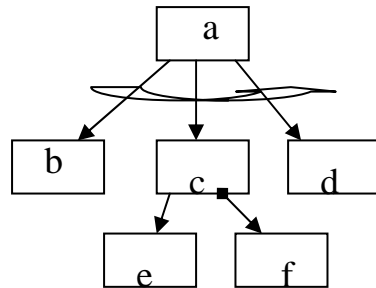
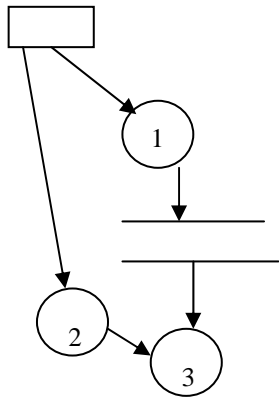
Object Oriented Metrics

**Excerpt from Guilherme Travassos' slides
University of Rio de Janeiro**

Product Measurement

Engineering Conventional Systems:

- functional decomposition (structured development) and some possible metrics



```

Program a
do while X
call b;
call c;
call d
end while;
    
```

```

Subroutine c
call e;
if y then call f
end;
    
```

Specification

Function Points
pages of documentation
estimated lines of code
calendar time
...

Design

Cohesion
Coupling
Complexity
Number of Modules
...

Implementation

Cyclomatic complexity
Halstead's theory
lines of code
Data bindings
Span
...

Testing

Coverage
Reliability
...

Product Measurement

Object Oriented Paradigm:

- means to organize the software as a collection of discrete objects that incorporate both data structure and behavior
- characteristics:

identity

abstraction

classification

encapsulation

inheritance

polymorphism

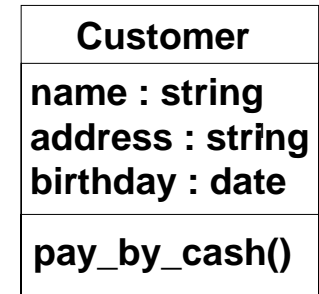
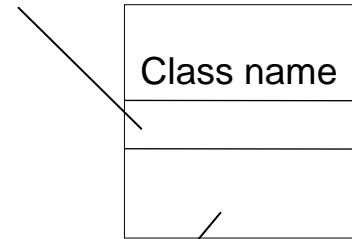
persistence

High-Level Design

UML Class diagram

- captures problem vocabulary
- relates everything together
- models information in a system
- anchors behavior to classes
- generates class structure declarations

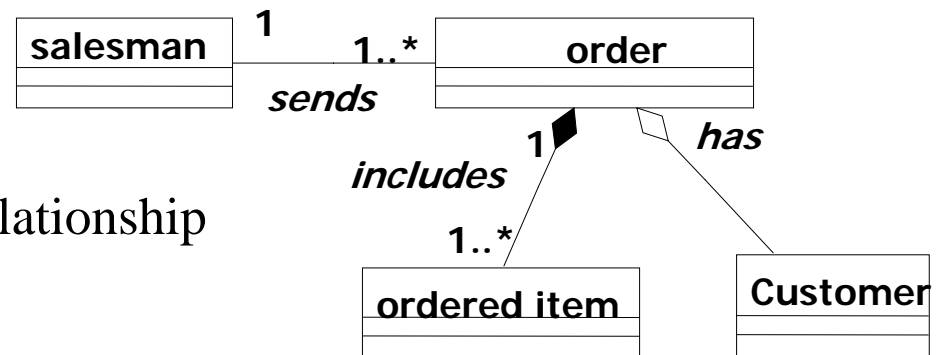
Attribute:Type = initial Value



operation(arg list):return type

UML Relationships

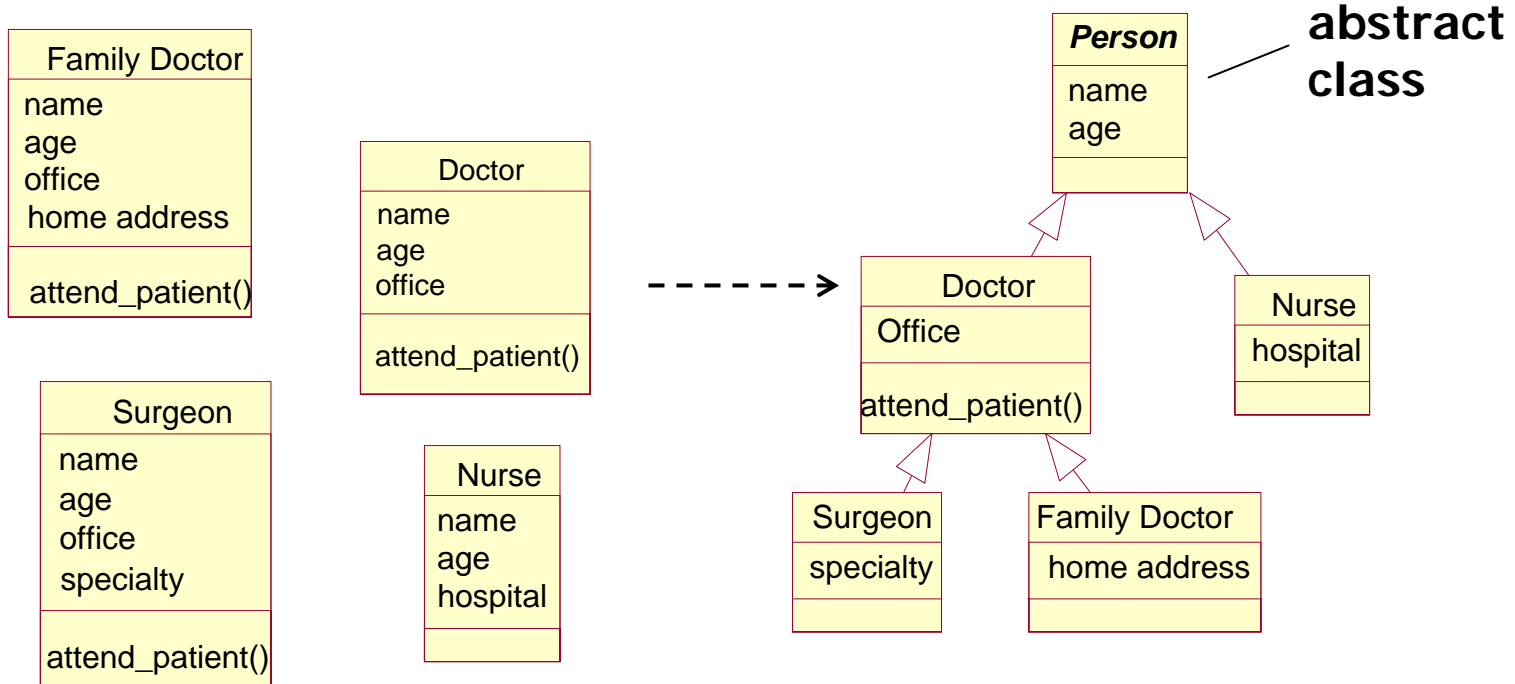
- association
 - aggregation: is the *part-of* relationship
 - composition: the whole

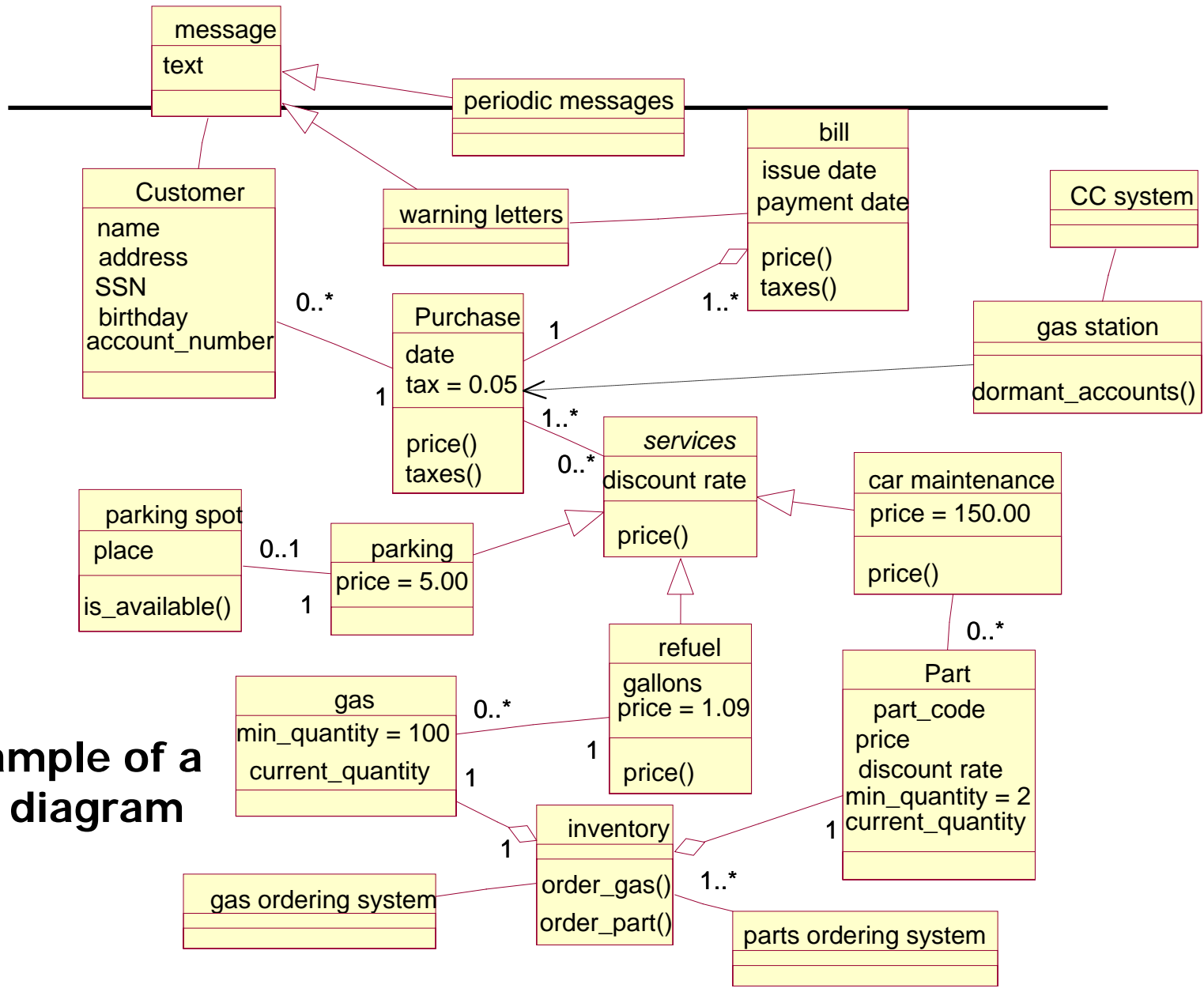


Object-Oriented Paradigm

class hierarchy using single inheritance

Structure identification:





An example of a class diagram

High-Level Design

UML Use Cases

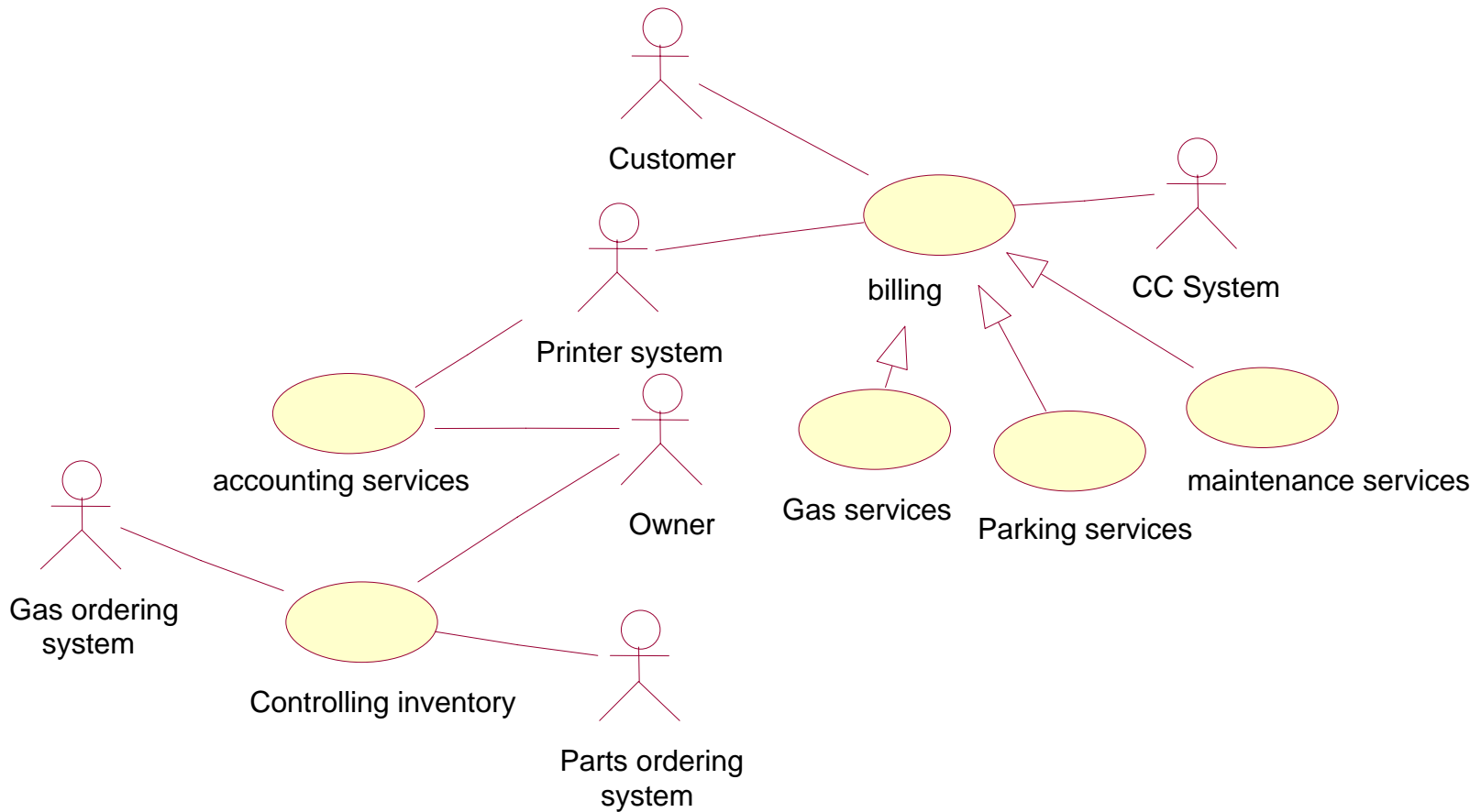
- capture some participant-visible function
- can be used to represent system's utilization contexts
- show some of the system requirements using high abstraction level

Diagram elements:

- **actor:** is a role that a participating plays to the system
- **use-case:** shows the visible system functionality
- **extensions:** extends a use-case
- **uses:** reuse of use-cases

High-Level Design

The Gas Station Use-Cases



Representing OO Software Product using *Unified Modeling Language - UML*

OO Design Artifacts

Dynamic View

- *use cases*
- activities
- interaction
 - sequences
 - collaborations
- state machines

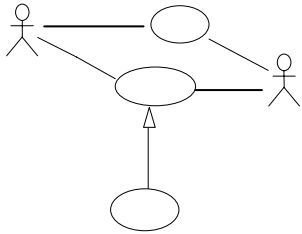
Static View

- classes
 - Relationships
 - Association
 - Generalization
 - Dependency
 - Realization
 - Extensibility
 - Constraints
 - Tagged values
 - Stereotypes

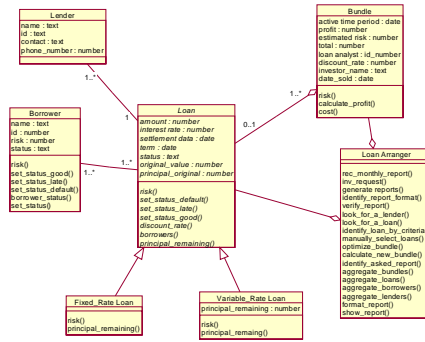
UML Diagrams

Representing OO Software Product using *Unified Modeling Language – UML*

...
 A gas station owner can use a system to control inventory. The system will either warn of low inventory or automatically order new parts and gas.



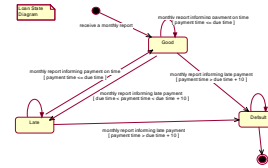
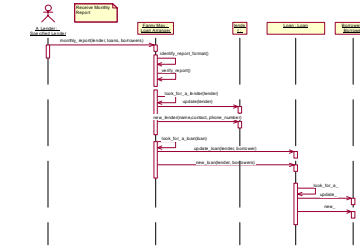
Requirements Description and Use Cases



Loan Arranger Classes Description

Class name: Fixed_Rate_Loan
Category: Logical View
Documentation:
 A fixed rate loan has the same interest rate over the entire term of the mortgage
External Documents:
Export Control: Public
Cardinality: n
Hierarchy:
 Superclasses: Loan
Public Interface:
Operations:
 risk
 principal_remaining
State machine: No
Concurrency: Sequential
Persistence: Persistent
Operation name: risk
Public member of: Fixed_Rate_Loan
Return Class: float
Documentation:
 take the average of the risks' sum of all borrowers related to this loan if the average risk is less than 1 round up to 1 else if the average risk is less than 100 round up to the nearest integer otherwise round down to 100
Concurrency: Sequential

High and Low Level Design



class parts
 inherit from **Stock_items**;
 attributes ...
 services
 relationships ...

Coding and Testing

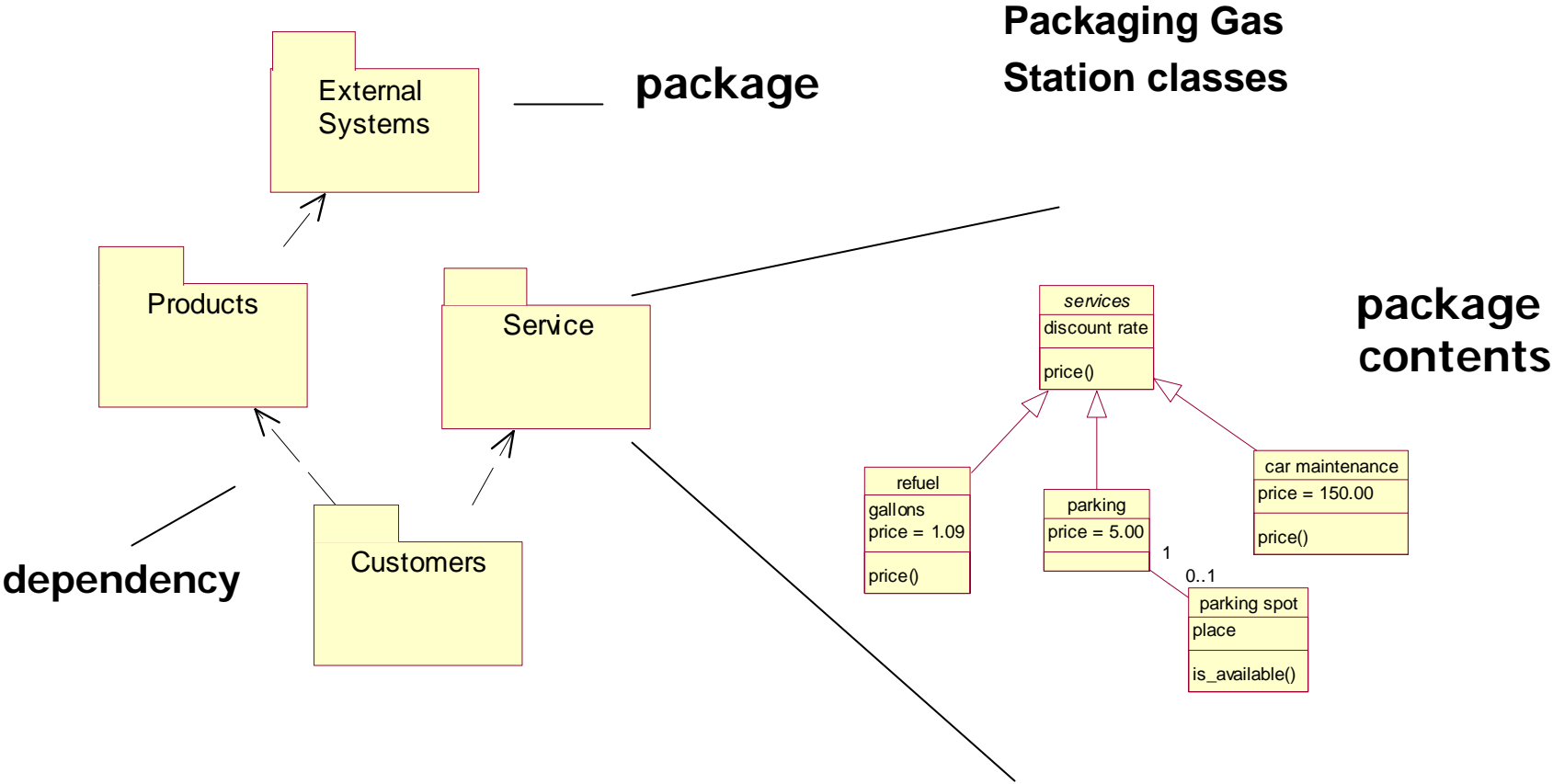
High-Level Design

UML Package diagrams

- used to break down a large system into smaller and meaningful sets of classes (clusters)
- shows the dependency among classes of different packages. A dependency exists between two elements if changes to the definition of one element may cause changes to the other (coupling)
 - Possible dependencies between two classes:
 - one class sends a message to other one
 - one class has other one as part of its data
 - one class mentions other one to be a parameter to an operation
- packaging is a vital technique for large projects:
 - allows to keep system's dependencies to a minimum reducing coupling
 - is particularly useful for testing

High-Level Design

UML Package diagram



Object-Oriented Software Measurement

Chidamber and Kemerer Metrics Suite [Chidamber94]

- Weighted Methods per Class (WMC)
- Depth of Inheritance (DIT)
- Number of children (NOC)
- Coupling Between Objects (CBO)
- Response for a class (RFC)
- Lack of Cohesion in Methods (LCOM)

Object-Oriented Software Measurement

Metric: **Weighted Methods per Class (WMC)**

Consider a Class C_1 , with methods M_1, \dots, M_n that are defined in the class. Let c_1, \dots, c_n be the complexity of the methods. Then:

$$WMC = \sum_{i=1}^n C_i$$

If all method complexities are considered to be unity, then $WMC = n$, the number of methods.

Viewpoints:

- 1) number of methods and the complexity of methods correlated with how much time and effort is required to develop and maintain the class
- 2) the larger the number of methods in a class the greater the potential impact on children, since children
- 3) Classes with large numbers of methods are likely to be more application specific, limiting the possibility of reuse

Object-Oriented Software Measurement

Metric **Depth of Inheritance (DIT)**

Depth of inheritance of the class is the DIT metric for the class. In cases involving multiple inheritance, the DIT will be the maximum length from node to the root of the tree.

Viewpoints:

- 1) The deeper a class is in the hierarchy, the greater the number of methods it is likely to inherit, making it more complex to predict its behavior
- 2) Deeper trees constitute greater design complexity, since more methods and classes are involved
- 3) The deeper a particular class is in the hierarchy, the greater the potential reuse of inherited methods

Object-Oriented Software Measurement

Metric: **Number of children (NOC)**

NOC = number of immediate subclasses subordinated to a class in the class hierarchy

Viewpoints:

- 1) Greater the number of children, greater the reuse, since inheritance is a form of reuse
- 2) Greater the number of children, the greater the likelihood of improper abstraction of the parent class. If a class has a large number of children, it may be a case of misuse of subclassing
- 3) The number of children gives an idea of the potential influence a class has on the design. If a class has a large number of children, it may require more testing of the methods in that class.

Object-Oriented Software Measurement

Metrics: **Coupling Between Objects (CBO)**

CBO for a class is a count of the number of other classes to which is coupled

Viewpoints:

- 1) excessive coupling between object classes is detrimental to modular design and prevents reuse. The more independent a class is, the easier it is to reuse it in another application
- 2) In order to improve modularity and promote encapsulation, inter-object class couples should be kept to a minimum. The larger the number of couples, the higher the sensitivity to changes in other parts of the design, and therefore maintenance is more difficult
- 3) A measure of coupling is useful to determine how complex the testing of various parts of a design are likely to be. The higher the inter-object class coupling, the more rigorous the testing needs to be.

Object-Oriented Software Measurement

Metric: **Response for a class (RFC)**

The response set of a class is a set of methods that can potentially be executed in response to a message received by an object of that class

$RFC = |RS|$ where RS is the response for the class.

$RS = \{M\} \cup \bigcup I \{R_i\}$

where $\{R_i\}$ = set of methods called by method I and

$\{M\}$ = st of all methods in the class

Viewpoints:

- 1) If a large number of methods can be invoked in response to a message, the testing and debugging of the class becomes more complicated since it requires a greater level of understanding required on the part of the tester
- 2) the larger the number of methods that can be invoked from a class, the greater the complexity of the class
- 3) A worst case value for possible responses will assist in appropriate allocation of testing time

Object-Oriented Software Measurement

Metric: **Lack of Cohesion in Methods (LCOM)**

Consider a class C1 with n methods M_1, M_2, \dots, M_n . Let $\{I_j\}$ = set of instance variables used by method M_j .

There are n such sets $\{I_1, \dots, I_n\}$. Let $P = \{(I_i, I_j) \mid I_i \cap I_j = \emptyset\}$

and $Q = \{(I_i, I_j) \mid I_i \cap I_j \neq \emptyset\}$. If all n sets $\{I_1, \dots, I_n\}$ are \emptyset then let $P = \emptyset$.

$LCOM = |P| - |Q|$, if $|P| > |Q|$

$LCOM = 0$ otherwise

Viewpoints:

- 1) Cohesiveness of methods within a class is desirable, since it promotes encapsulation
- 2) Lack of cohesion implies classes should be probably be split into two or more subclasses
- 3) Any measure of disparateness of methods helps identify flaws in the design of classes
- 4) Low cohesion increases complexity, thereby increasing the likelihood of errors during the development process.

Object-Oriented Software Measurement

Metrics x Process

Phase Metric	Requirements Description	High Level Design	Low Level Design	Coding	Testing
WMC					
DIT					
NOC					
CBO					
RFC					
LCOM					

Object-Oriented Software Measurement

Validation

Basili, Briand and Melo [Basili95]

- Based on CK suite, empirically validate the suitability of such metrics for predicting the probability of detecting faulty classes for C++ based software
- Slightly adjusted some of the metrics to capture C++ flavors (the metrics are not language independent)
- WMC:
 - all methods have complexity 1
 - “friend” operators are not counted

Object-Oriented Design

Bibliography

- Fowler97. Martin Fowler and Kendall Scott, UML Distilled: Applying the standard object modeling language, Addison-Wesley, 1997
- Booch94. Grady Booch, Object-Oriented Analysis and Design with Applications, The Benjamin/Cummings Publishing Company, Inc, second edition, 1994
- Lorenz94. Lorenz, M., and J. Kidd, Object-Oriented Software Metrics, Prentice-Hall, 1994
- Chidamber94. Chidamber, S.R. and Kemerer, C. F.; A Metrics Suite for Object Oriented Design, IEEE Transactions on Software Engineering, vol.20, no6, June 1994.
- Basili95. Basili, Victor R.; Briand, Lionel and Melo, Walcelio; A validation of Object-Oriented design metrics, Communications of the ACM, XXX