

UT Dallas

Software Quality and Software Testing

Part 1 – The Big Picture (How Quality Relates to Testing)

Part 2 – Fundamental Concepts of Measurement and Data Analysis

Part 3 – Defect Containment

Part 4 – Measuring Software Structure

Part 5 – Measuring Software Complexity

Dennis J. Frailey

Retired Principal Fellow - Raytheon Company

PhD Purdue, 1971, Computer Science

Assistant Professor, SMU, 1970-75

Associate Professor, SMU, 1975-77

(various titles), Texas Instruments, 1974-1997;

Raytheon Co. 1997-2010

Adjunct Associate Professor, UT Austin, 1981-86

Adjunct Professor, SMU, 1987-2019

Adjunct Professor, UT Arlington, 2014-present

**Areas of specialty: software development
process, software project management,
software quality engineering, software metrics,
compiler design, operating system design, real-
time system design, computer architecture**

Part 3

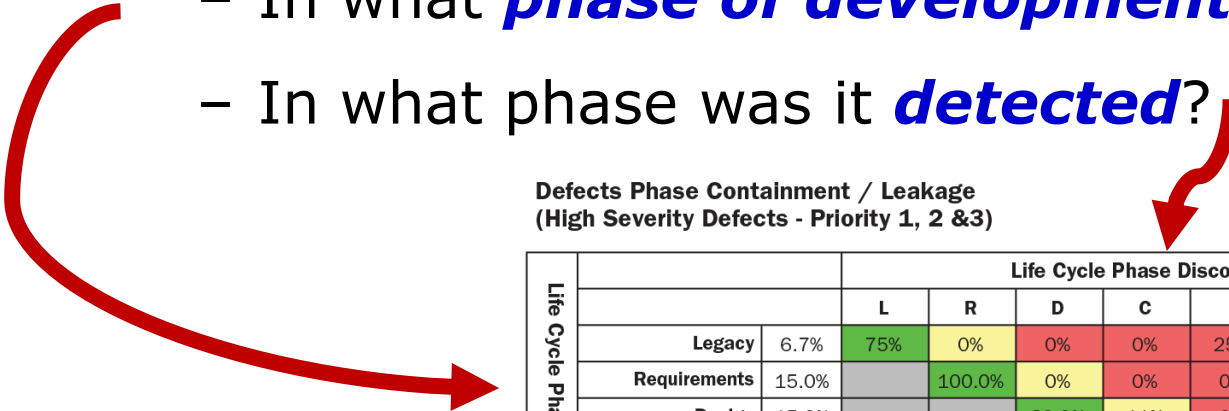
Defect Containment (Phase Containment)

Defect Containment (Phase Containment)

This requires that you collect additional information about each defect you discover during an inspection or as a result of a test:

- In what *phase of development* was the defect *created*?
- In what phase was it *detected*?

Defects Phase Containment / Leakage
(High Severity Defects - Priority 1, 2 & 3)



Life Cycle Phase Originated			Life Cycle Phase Discovered							In the red
			L	R	D	C	I	T	A	
	Legacy	6.7%	75%	0%	0%	0%	25%	0%	0%	
	Requirements	15.0%		100.0%	0%	0%	0%	0%	0%	
	Design	15.0%			89.9%	11%	0%	0%	0%	
	Code and Unit Test	63.3%				94.7%	5.3%	0%	0%	
	Integration Test	0.0%					0%	0%	0%	
	Test									
	After Test									

Insights.sei.cmu.edu

Note on Defect Containment

- **There are several variations on this method**
- **All use the same basic data (base measures) but they use the data in different ways**

**In this course we will illustrate one of the variations on this method.
You may find others at
www.sei.cmu.edu**

Example of Defect Containment

- Suppose you ***detect*** a lot of defects during ***system test***
- And suppose you discover that most of them ***occurred*** due to ***bad design procedures***
- Then you know that the best way to fix the problem is to ***improve your design procedures***

In-Phase Defects

In-phase defects are those that are ***corrected in the same development phase*** where they were introduced

- Example: a coding error that is caught and corrected while you are writing the code, before going to system test
- **Measuring in-phase defects tells you which parts of your process generate large numbers of defects**

In-phase defects are generally the least costly to correct.

Out-of-Phase (Leaking) Defects

Out-of-phase defects are those that are detected (and corrected) ***after they leave*** the phase where they were introduced

- Example: a design error caught during unit test
- **Measuring out-of-phase defects indicates how often you allow defects to “leak” from the phase where they originate**
 - **this is a predictor of post-release failures**
 - and also a good help in ***root cause analysis***

Finding the
Ultimate Cause
of a Defect

Out-of-phase defects are generally
the most costly to correct.

Defect Containment Analysis

Step 1 – Collect the Data

Track Each Defect and Record Phase of Origin

Defect Report

Description _____

Phase where found _____

Phase where introduced _____

Priority _____ **Type** _____

Estimated Cost to Fix _____

etc.

Defect Containment Analysis

Step 2 – Record and Display the Data

Defect Containment Matrix – Sequential Process

		Phase where Defect was Inserted					
		RA	PD	DD	C&T	I&T	POST REL.
Phase where Defect was Detected	RA	15					
	PD	12	55				
	DD	42	8	23			
	C&T	15	3	8	17		
	I&T						
	POST REL.						

This shows the data at the end of the C&T phase

Defect Containment Analysis

Step 2 – Record and Display the Data

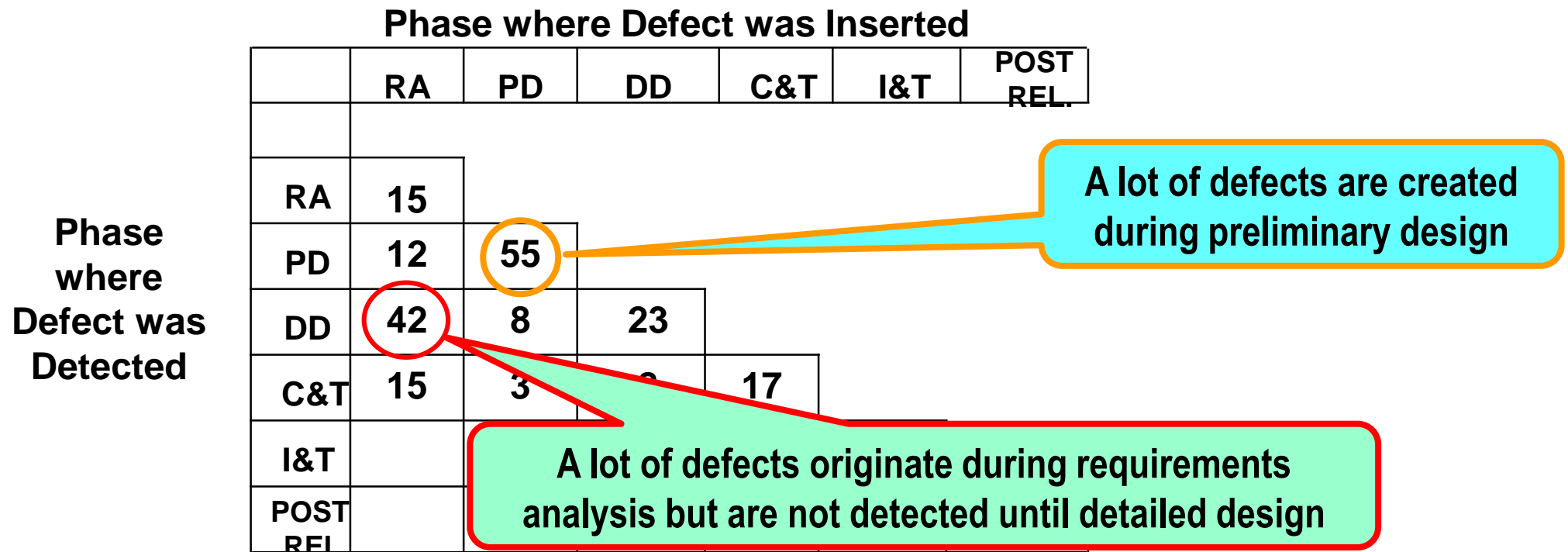
Defect Containment Matrix – SCRUM Process

		Scrum where Defect was Inserted					
		S1	S2	S3	S4	S5	POST REL.
Scrum where Defect was Detected							
	S1	15					
	S2	12	55				
	S3	42	8	23			
	S4	15	3	8	17		
	S5						
		POST REL.					

This shows the data at the end of the 4th SCRUM

Defect Containment Analysis Step 3 - Using the Data

If you see many out-of-phase defects in a specific cell, you can narrow down the source of defects



Defect Containment Analysis Step 4 - Using the Data to Provide Additional Insight

Over time, you can correlate

- **the number of defects in the matrix**
- **to the number of failures found by the customer**
- **You can use this to predict and ultimately to manage the number of failures**

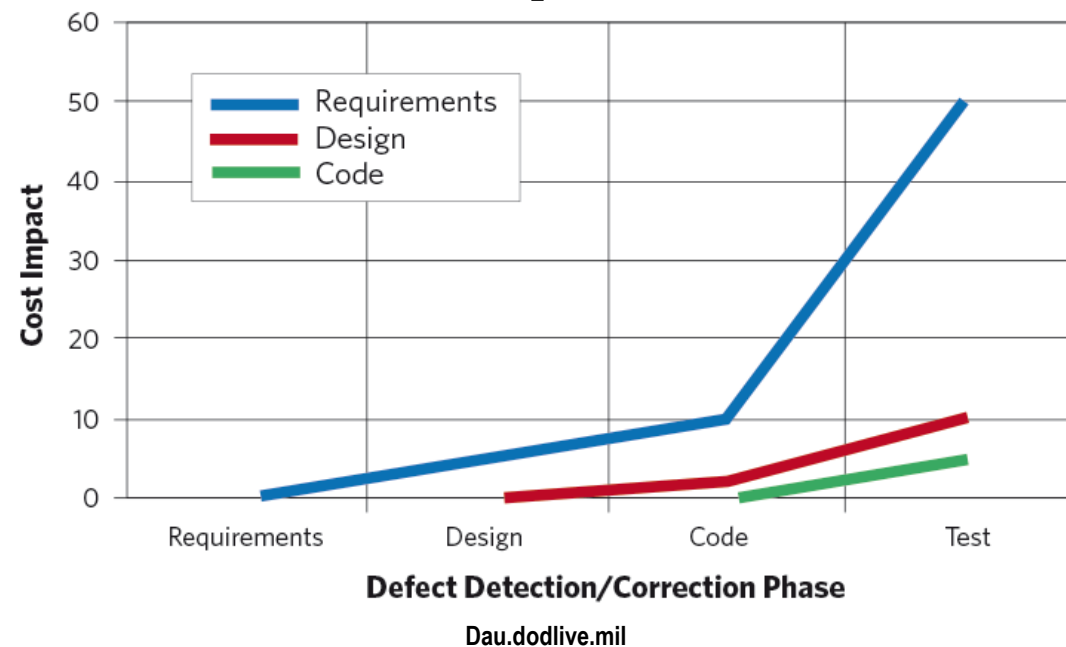
A method for doing this will be shown briefly in today's lecture

- **Definition of a defect must be adhered to in a consistent way across the project**
- **As shown, there is no distinction by type or severity of defect**

(but this distinction can also be made if the original data are good enough)

A Key Lesson Learned from Measuring Defect Containment

If you detect and correct defects early, it greatly reduces cost and reduces post-release failures (i.e., those seen by the customer)



- **But this requires very good understanding of requirements and of customer “care-about”**

Contained and Leaking Defects

Phase of Injection

Phase of Detection		RA	PD	DD	C&UT	I&T	Post Rel
	RA	15					
	PD	12	55				
	DD	22	8	23			
	C&UT	15	3	8	17		
	I&T						
	Post Rel						

In-phase or Contained

Out-of-phase or Leaking

Large Numbers Indicate Software Development Process Problems

- **Large numbers in any column indicate that your development process is generating many defects in that process phase**
- **A large number in a “leaking” cell means you are also paying a lot of money for rework**

This tells you where to focus process improvement efforts

A Typical Defect Containment Chart

Phase Detected	Phase Originated							total
	RA	PD	DD	CUT	I&T	SYS INT	POST REL	
RA	730							730
PD	158	481						639
DD	19	2	501					522
CUT	15	0	12	63				90
I&T	25	4	35	321	9			394
SYS INT	4	0	7	19	4	2		36
POST REL	48	2	0	36	0	0	67	153
total	999	489	555	439	13	2	67	2565

Least Costly Defects are on the Diagonal

These defects are "Contained" within the step where they were caused

Escaping Defects are Those Not Detected until After Release

Phase Originated								
Phase Detected	RA	PD	DD	CUT	I&T	SYS INT	POST REL	total
RA	730							730
PD	158	481						639
DD	19	2	501					522
CUT	15	0	12	63				90
I&T	25	4	35	321	9			394
SYS INT	4	0	7	19	4	2		36
POST REL	48	2	0	36	0	0	67	153
total	999	489	555	439	13	2	67	2564

Escaping Defects Cost the Most of All

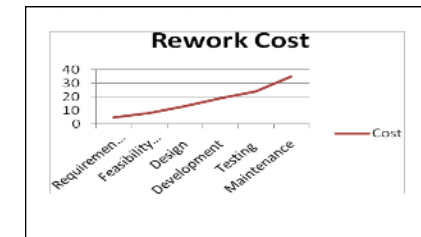
Other Uses of Defect Containment Data

There are many uses of defect containment

- Calculating ***total repair cost***
 - By recording labor cost to repair defects
- Calculating ***rework cost***
 - Reduction in rework can be compared with cost of prevention activities
- ***Organizational-level*** analysis
- ***Prediction*** of ***defects*** and ***warranty costs***
- ***Prediction*** of ***reliability***



Aspennw.com



ljser.org



Sciencedirect.com



Defect Repair Cost

Labor Cost to Repair Defects

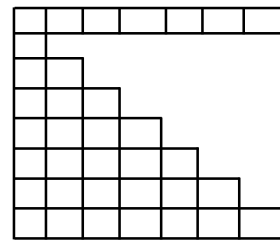
Phase of Injection

Phase of Detection		RA	PD	DD	C&UT	I&T	Post Rel
	RA	\$1					
	PD	\$12	\$2				
	DD	\$22	\$8	\$2			
	C&UT	\$45	\$18	\$8	\$2		
	I&T						
	Post Rel						

Cell i,j indicates the average labor cost to **repair a defect** *created* in phase i and *detected* in phase j



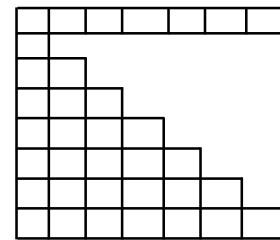
If you multiply the defect containment chart by the “labor cost to repair” chart, you get total repair cost



Defect
Counts

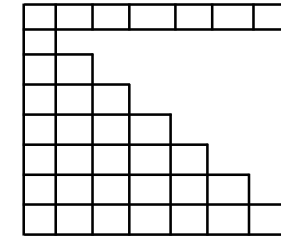


Cell-wise
multiplication



Cost to
Repair

=



Total
Repair Cost

Total Repair Cost Example

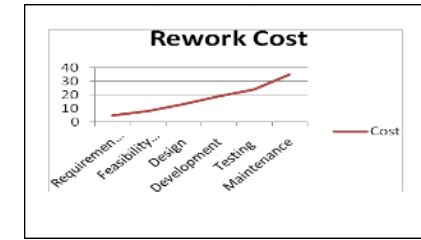


Aspennw.com

		Phase of Injection					
		RA	PD	DD	C&UT	I&T	Post Rel
Phase of Detection	RA	\$15					
	PD	\$144	\$110				
	DD	\$484	\$64	\$46			
	C&UT	\$675	\$54	\$64	\$34		
	I&T						
	Post Rel						

Cell i,j indicates the total labor cost to repair all defects created in phase i and detected in phase j

Rework Costs Are The Portion Of the Prior Chart That Are Not On The Diagonal



ljser.org

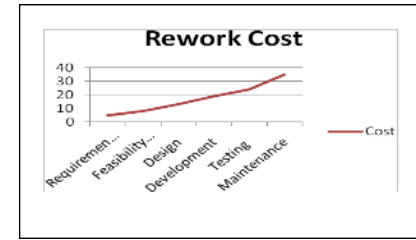
Phase of Injection

	RA	PD	DD	C&UT	I&T	Post Rel
Phase of Detection						
RA	\$15					
PD	\$144	\$110				
DD	\$484	\$64	\$46			
C&UT	\$675	\$54	\$64	\$34		
I&T						
Post Rel						

Costs off-diagonal are rework costs

This Concept Applies Throughout the Product Lifetime

You can *track repair cost* and *rework cost* during development and after delivery to the customer



ljser.org



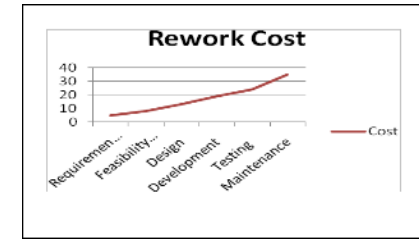
Aspennw.com

- You can further *break defects down by characteristics*:
 - Phase of Development where Defect Occurred
 - Severity
 - Importance to Customer
 - Cost to Repair
 - Time to Repair
 - Which Part of the Software was Responsible
 - Etc.



imgkid.com

This Can Help You Justify Process Improvements



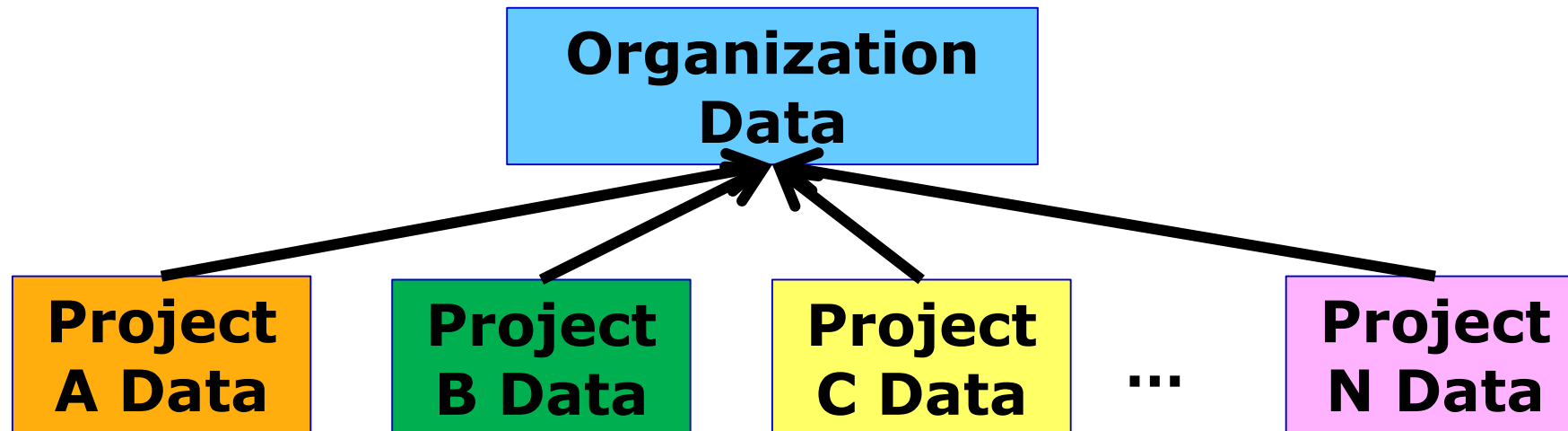
ljser.org

Rework costs are the equivalent of “software scrap”

- If you can *reduce scrap* by *investing in defect prevention* activities, you can save a lot of money (see earlier modules)
- If you make an improvement in your development process, you can *use the defect containment chart to show the savings* in reduced repair cost
- And you can use the chart to *determine which parts of the process are most important to improve*

Analyzing Defect Data at the Organizational Level

- By collecting data from many projects, we can show historical costs for rework
- And we can also show *patterns* of defect containment



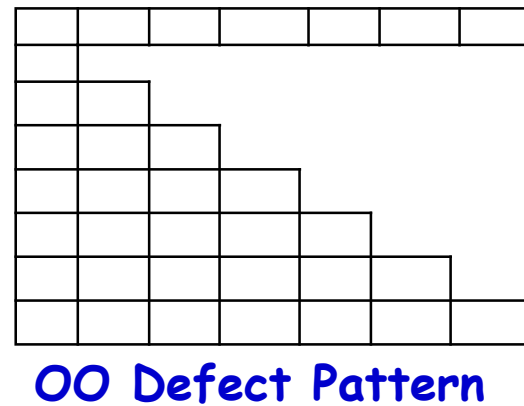
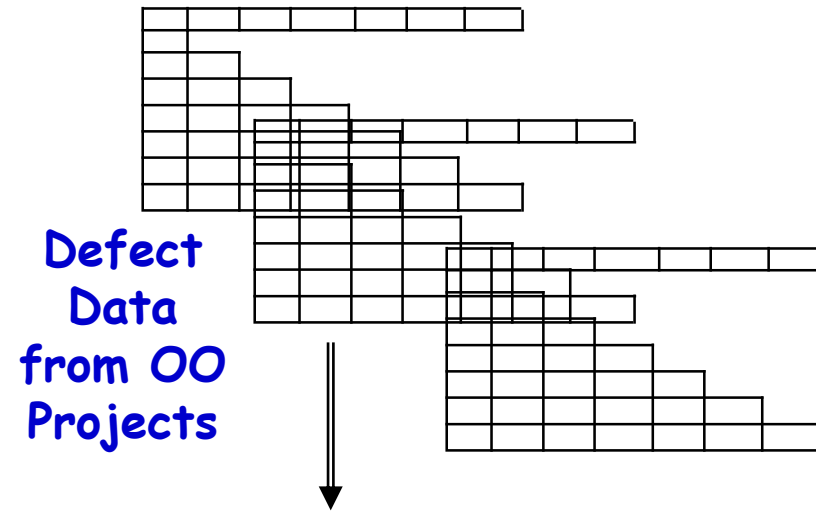
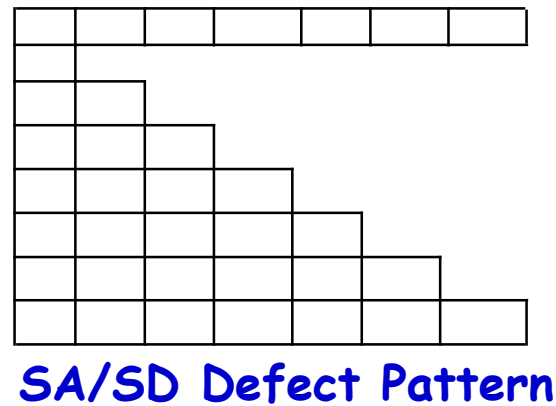
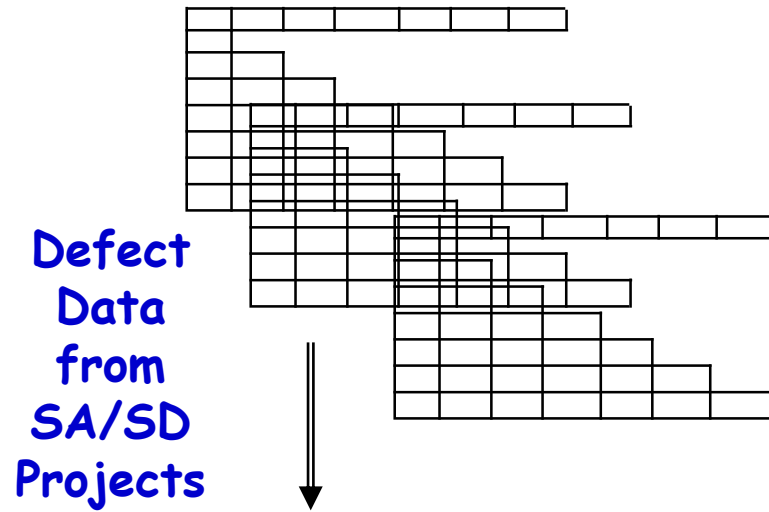
Organizational Analysis of Defect Containment Data

Analysis of defect containment data for many projects over a period of time may show such organizational information as:

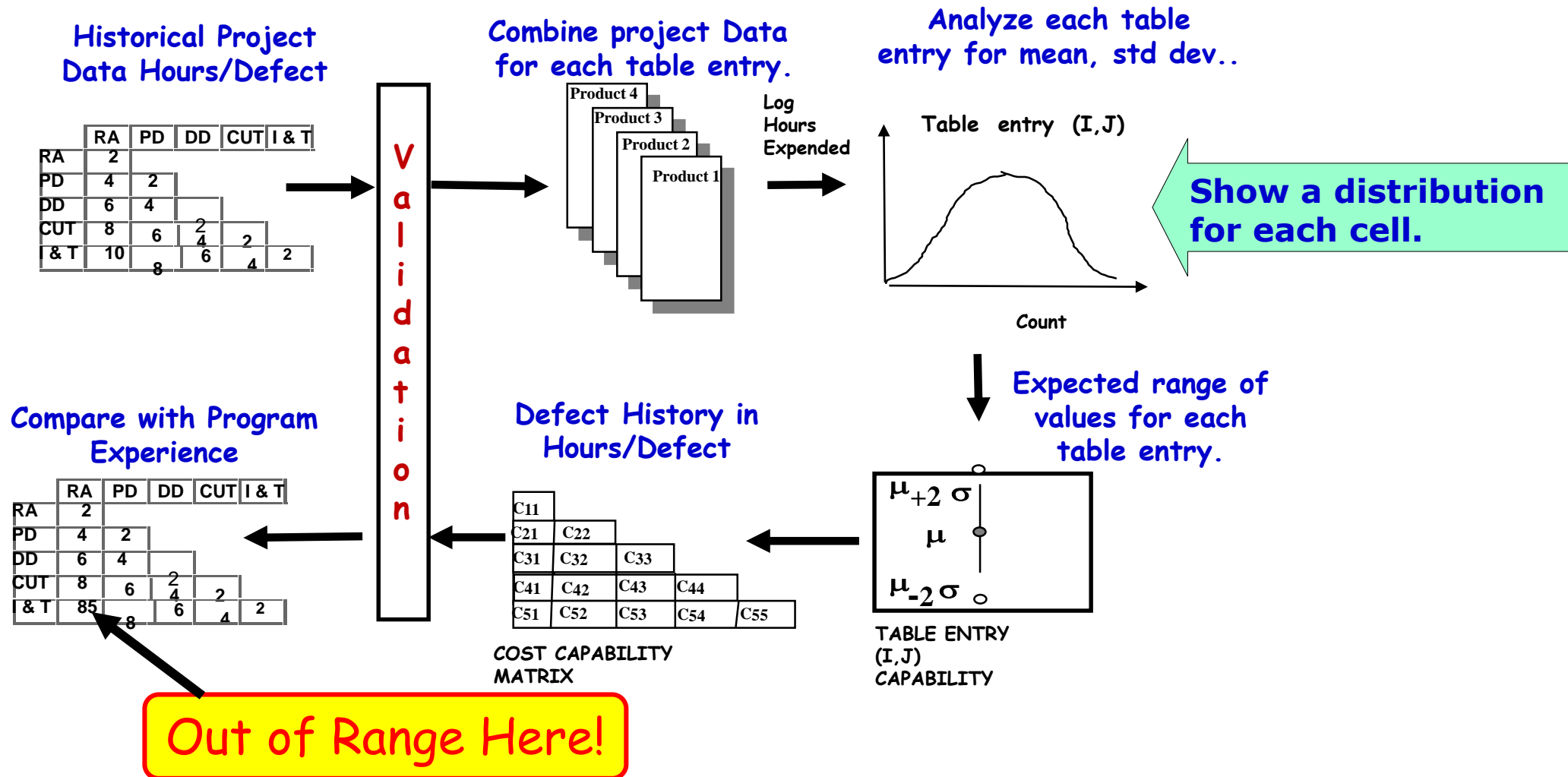
- ***Most frequent*** types of defects
- ***Most costly*** defects
- ***Time required to fix*** defects
- ***Process steps generating the most defects***
- ***Which design standards help or hurt defects***

**Typically we collect the data needed for statistical process control:
averages, ranges, distributions, maximum, minimum, etc.**

Example: Determining an Organizational Process Metric



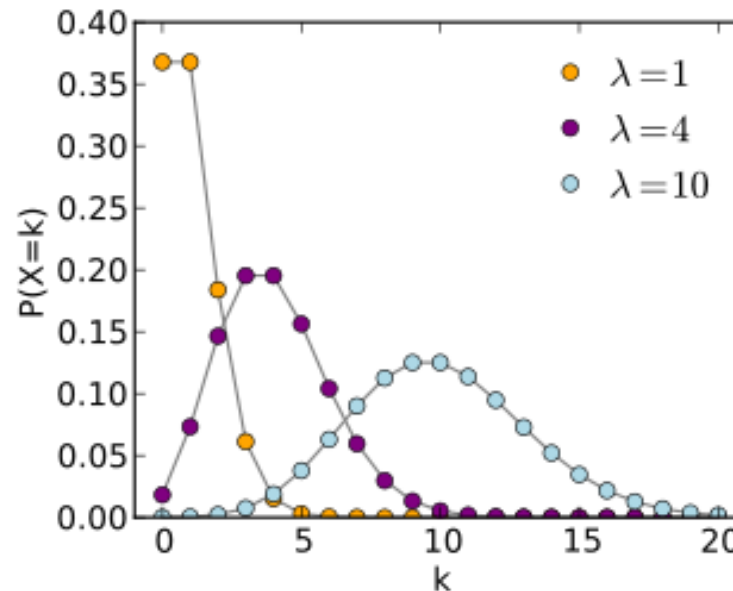
Overview of the Cost Data Collection Process for an Organization



UTD Before We Discuss Additional Uses of Defect Containment ...

We need to introduce a special distribution that we haven't seen before

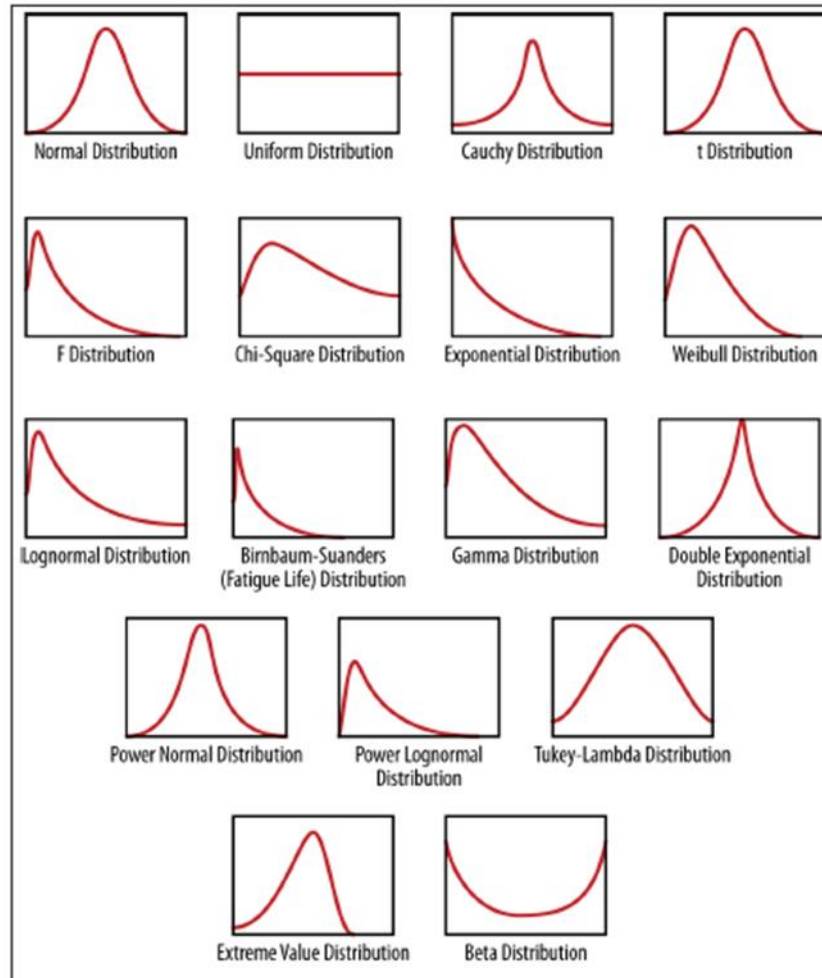
Poisson Distribution



Contents

- Applications of Defect Containment
- ***Poisson Distributions***
- A Model for Predicting Defect Levels and Associated Costs, Using Defect Containment
- Predicting Reliability
- Measuring the Cost of Defect Removal

Recall We Introduced Distributions in the Previous Lecture



We mentioned that there are many distributions that correlate to actual data.

Today we will introduce the Poisson distribution

Exponential Distribution Formula

You may be familiar with the exponential distribution function:

Exponential

$$f(t) = \lambda e^{-\lambda t}$$

Poisson Distribution Formula

Poisson distribution has a similar formula:

Exponential	$f(t) = \lambda e^{-\lambda t}$
Poisson	$f(t) = \lambda^t e^{-\lambda} / t!$

$t! = t$ factorial ($1*2*...*t$)

Poisson distribution is only defined for positive *integer* values of t
For Poisson distributions, λ is the average or mean value of t

Uses of Poisson Distribution

- Exponential and many other distributions are used for situations where the *independent variable (t) is continuous*
 - t can be any value or any non-negative value
 - such distributions are often used for estimating when an event will occur such as when a failure or defect will occur
- Poisson distribution is used for situations where the *independent variable (t) is a discrete, positive integer*
 - Often used for predicting the number of events (for example, number of failures or defects) that will occur in a given time period

Poisson Formula is Often Written Using the Letter **k** rather than **t**

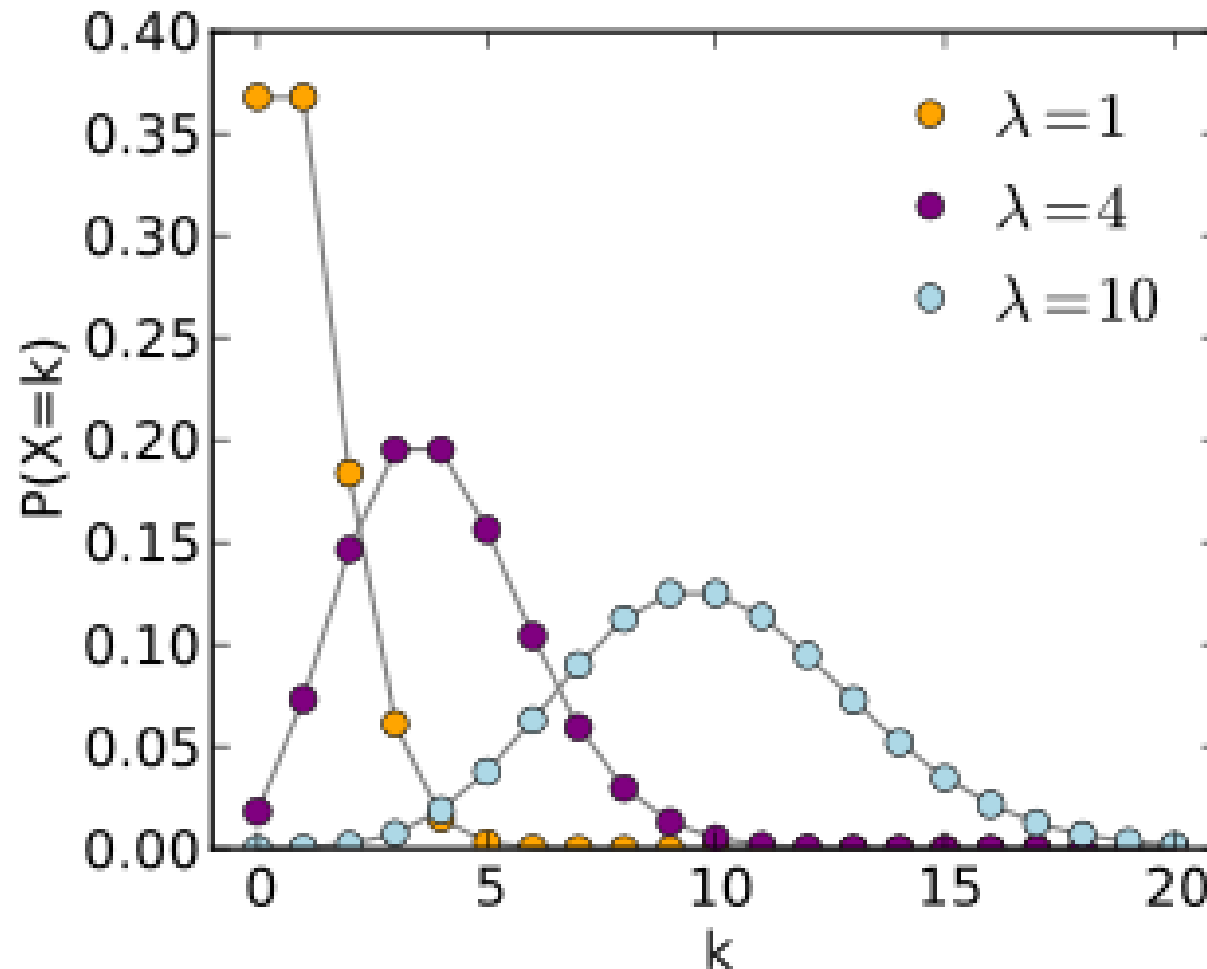
To suggest an integer value

$$f(k) = \lambda^k e^{-\lambda} / k!$$

k is a number of occurrences

f(k) is the probability that an event (such as a failure) will occur **k** times

Poisson Distributions



k (horizontal axis)
is number of
occurrences

$P(X=k)$ (vertical
axis) is the
probability that an
event will have
the indicated
number of
occurrences

Advantages of Poisson Distribution

Poisson distributions are often used for situations where occurrences are discrete, independent and relatively uncommon

Benefits:

- Relatively ***un-restrictive assumptions***
- Relatively ***straightforward derivation*** and a relatively simple model
- Among other things, **the mean = the variance**, which ***simplifies many calculations***

Thus Poisson distributions are widely used in software reliability modeling

A Practical Note about Calculating Poisson Distributions on a Computer

$$f(k) = \lambda^k e^{-\lambda} / k!$$

- If k is large, both λ^k and $k!$ may be *very large numbers*
 - This may lead to overflow or underflow and, thus, highly inaccurate/unstable calculations

This is a good example of a common situation on computers: what is simple mathematically may not be so simple to implement on a computer

A Mathematically Equivalent But Computationally More Stable Equation

On a computer, you may want to compute the Poisson distribution in this fashion:

$$f(k) = \exp\{k \ln \lambda - \lambda - \ln \Gamma(k+1)\}$$

Where Γ is the “gamma function”

In Excel:

GAMMA() is the gamma function: $\Gamma()$

GAMMALN() is the natural logarithm of the gamma function:

$\ln \Gamma()$

In MATLAB: gammaln()

In C standard library: lgamma()

Contents

- Applications of Defect Containment
- Poisson Distributions
- ***A Model for Predicting Defect Levels and Associated Costs, Using Defect Containment***
- Predicting Reliability
- Measuring the Cost of Defect Removal

We Can *Predict* Rework and Other Costs from Defect Level

- A *predictive model*, to be introduced in the next several slides, relates future defect levels to defect containment values

For a given project or group of similar projects, we can *predict future defects* and *rework cost*

as a function of

the defect level achieved during software development

- The model can be applied early in development and continually refined as development proceeds
 - So you can *spot potential trouble early*

Predicting Quality

A Model Based on Defect Containment

Assumption:

We have a software development process with **N** phases

L_i = the number of defects introduced in phase **i**

Other Assumptions:

1. **L_i** has a Poisson distribution with mean **λ_i**
2. **L_i** and **L_j** are independent for **i** **j**.

Continuation of the Model

K_{ij} = the number of defects detected in phase **j**
that originated in phase **i**

For additional information on this model, see
Hedstrom and Watson in the Reference List

The Model

Phase of Injection

Phase of
Detection

	RA	PD	DD	C&UT	I&T	
RA	K11					
PD	K12	K22				
DD	K13	K23	K33			
C&UT	K14	K24	K34	K44		
I&T	K15	K25	K35	K45	K55	
TOTAL	L1	L2	L3	L4	L5	

More of the Model

P_{ij} = the probability that a defect introduced in phase i will be detected in phase j

Assumptions:

The detection forms a Bernoulli process

[in other words, a software development process where the individual steps are independent from each other]

and

Detection of one defect is independent of others

Distribution of Detected Defects

It can be shown that K_{ij} has a Poisson distribution with mean $\lambda_i P_{ij}$. [Ross, 1993]

It can also be shown that the number of defects leaking from the phase where they were introduced, $L_i - K_{ij}$,

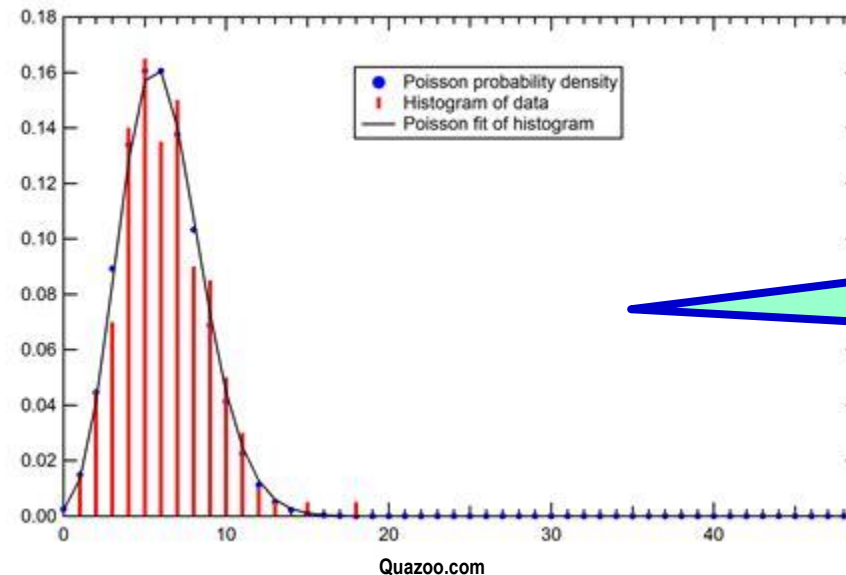
has a Poisson distribution with mean $\lambda_i(1 - P_{ij})$.

Reminder: A Poisson distribution is a common assumption when dealing with discrete, independent and relatively rare events. For more on this see Ross or Knuth in references.

In general, the number of phase **i** defects detected in the **j**th phase (i.e., **K_{ij}**)

has a Poisson distribution with mean:

$$(1-P_{ii}) * (1-P_{ii+1}) * \dots * (1-P_{ij-1}) * P_{ij} \lambda_i$$



It might look something like this

Distribution of Escaping Defects

Let $L_i^* = L_i - \sum_{j=1..N} K_{ij}$ be

the number of step **i** defects not detected by the **N**th step
of the process

(i.e., the number of escaping defects from step **i**)

Distribution of Escaping Defects

(continued)

It can be shown by induction that L_i^* has a Poisson distribution with mean:

$$\lambda_i^* = \lambda_i \prod (1 - P_{ij}) \quad [j=1..N]$$

which we can rewrite as:

$$\lambda_i^* = a_i \lambda_i$$

$$\text{where } a_i = \prod (1 - P_{ij}) \quad [j=1..N]$$

Total Escaping Defects

$$L^* = \sum L_i^* [i=1..N]$$

is the total number of escaping defects.

It has a Poisson distribution with mean:

$$\lambda^* = \sum \lambda_i^* = \sum \lambda_i \Pi (1-P_{ij}) = \sum a_i \lambda_i [i=1..N]$$

Leaking and Escaping Defects

	RA	PD	DD	C&UT	I&T	
RA	K11					
PD	K12	K22				
DD	K13	K23	K33			
C&UT	K14	K24	K34	K44		
I&T	K15	K25	K35	K45	K55	
TOTAL	L1	L2	L3	L4	L5	
Escaping	L1*	L2*	L3*	L4*	L5*	Sum = L*

Leaking (indicated by a green arrow pointing to the diagonal cells K11 through K55)

Escaping (indicated by a green arrow pointing to the row L1* through L5*)

Total Detected Defects

It can also be shown that $K_i = \sum_{j=1..N} K_{ij}$,
[the total number of defects from stage i which were detected]

has a Poisson distribution with mean and variance given by:

$$\text{Mean } (K_i) = a_i \lambda_i \sum_{j=1..N} (P_{ij} / b_{ij})$$

$$\text{Var } (K_i) = a_i \lambda_i \sum_{j=1..N} (P_{ij} / b_{ij})$$

$$\text{where } b_{ij} = \prod_{k=j..N} (1 - P_{ik})$$

(mean = variance because Poisson)

Estimating Escaping Defects

The K_i form a sufficient statistic for estimating the λ_i and hence the λ_i^* and λ^* .

Maximum likelihood estimators for the λ_i are given in [*Hedstrom and Watson, 1995*], along with additional details of the model and its derivation.

The point is that we can estimate escaping defects as a function of measurable data, namely the K_i and the P_{ij} .

The numbers in the defect containment matrix

Uses of the Model

Defect Predictive Engine

- The ***defect predictive engine*** is a spreadsheet that
 - uses the formulas on the previous slides
 - to relate the final escaping defect level
 - to the numbers in the defect containment chart
- The engine ***predicts final defect level***, given estimated or actual defect levels

Stage Detected	Stage Originated							Total
	Requirements	Design	Code and Unit Test	SW Integration	SW Quality Test	System Integration and test	SW Maintenance	
Requirements	1,515							1,515
Design	1,181	1,555						2,736
Code and Unit Test	402	912	2,421					3,735
SW* Integration	200	420	1,525	37				2,182
SW Quality Test	191	223	370	7	1			792
System Integration and Test	89	114	114	5	0	10		332
SW Maintenance	0	0	0	0	0	0	0	0
Total	3,578	3,224	4,430	49	1	10	0	11,292

Semanticscholar.org



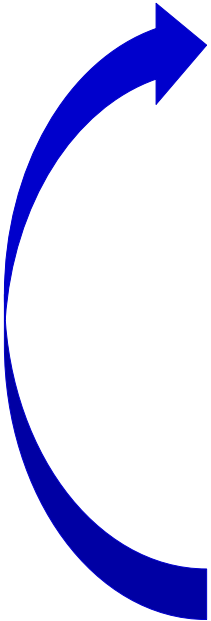
Dreamstime.com

You Can Manage the Process to Achieve a Desired Defect Level

1. Start with historical data

- Populate the defect containment chart with historical averages from your organization
- Predict the escaping defects, using the formulas in the model

2. Augment with actual data

- 
- As the development project proceeds, replace the averages with actual project data
 - Recompute the predicted escaping defect levels
 - If the levels are too high, refine the process, such as:
 - More rigorous inspections, testing and peer reviews
 - More careful development practices
 - Etc.
 - Repeat

Example: Quantifying the Warranty Cost for a Software Product (1 of 3)

Consider a software product with a predicted number of defects.

	Require-ments	Prelim. Design	Detailed Design	Code & Unit Test	Integ. & Test
Predicted Defects Escaping	16	19	16	16	1

Example: Quantifying the Warranty Cost for a Software Product (2 of 3)

Consider a software product with a predicted number of defects.

- **Estimate the potential cost of warranty work (labor hours * \$xx/hr.).**

	Require-ments	Prelim. Design	Detailed Design	Code & Unit Test	Integ. & Test
Predicted Defects Escaping	16	19	16	16	1
Predicted Repair Cost (\$K)	\$23.4	\$47.9	\$27.1	\$22.5	\$30.0

Example: Quantifying the Warranty Cost for a Software Product (3 of 3)

Consider a software product with a predicted number of defects.

- Estimate the potential cost of warranty work (labor hours * \$xx/hr.).

	Require-ments	Prelim. Design	Detailed Design	Code & Unit Test	Integ. & Test	Estimated Warranty Cost
Predicted Defects Escaping	16	19	16	16	1	↓
Predicted Repair Cost (\$K)	\$23.4	\$47.9	\$27.1	\$22.5	\$30.0	\$150.9

Total the warranty estimates

Other Ways to Use the Information from the Model and the Engine

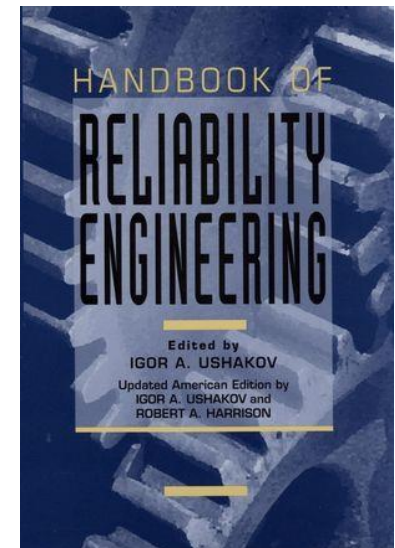
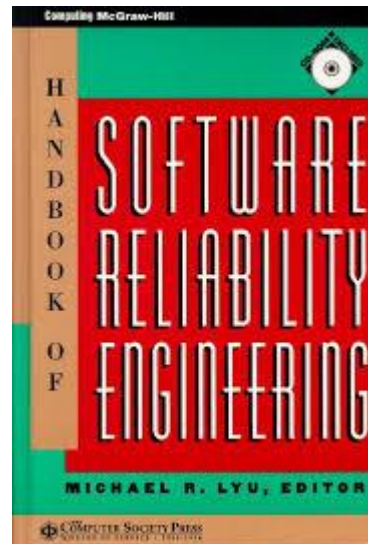
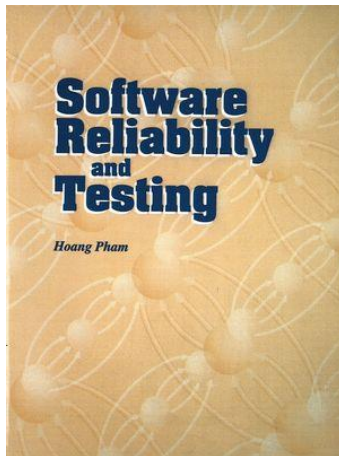
- **Improve defect detection**
 - Model/Engine shows where detection is needed and how it will pay off in two ways:
 - Reduced rework during development
 - Reduced warranty cost
- **Reduce defects by fixing the software development process**
 - Model/Engine pinpoints where the problems are coming from and the potential level of pay off

Contents

- Applications of Defect Containment
- Poisson Distributions
- A Model for Predicting Defect Levels and Associated Costs, Using Defect Containment
- **Predicting Reliability**
- Measuring the Cost of Defect Removal

Defects and Reliability are Related, Although Not Always Strongly

- As we mentioned earlier in the course, the number of defects may or may not correlate well with reliability
- Many studies do show that reliability is related to defect levels in practice



Capers Jones' Data

Defects per KLOC	Reliability (MTTF)
>20	2-15 minutes
10-20	5-60 minutes
5-10	1-4 hours
2-5	4-24 hours
1-2	24-160 hours

See references.
Assembly language code written in the 1970's.

Escaping Defects and Reliability

Musa [see references] has shown a simple reliability model that requires an estimate of escaping defects.

$$\lambda_0 = fK\omega_0$$

λ_0 is the failure rate

f is the rate at which the software is used

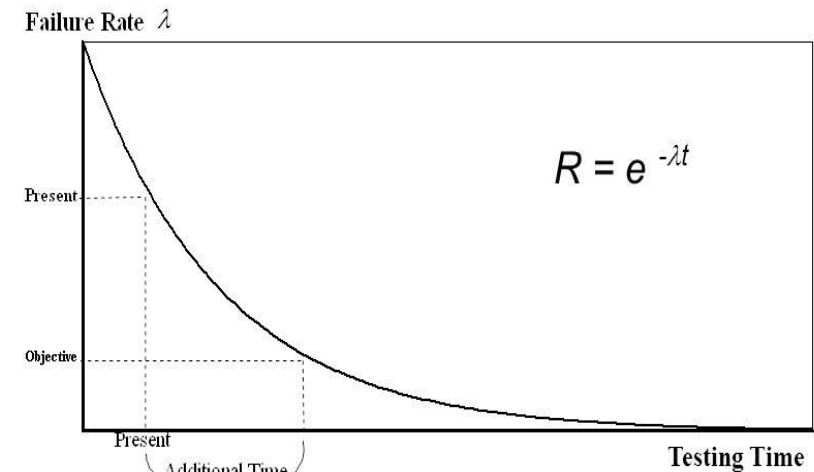
K is a small constant indicating which portion of faults are visible as failures

ω_0 is the escaping defect level.

So we could use the defect containment model to predict reliability.

- **Capture data to correlate defects with reliability or repair costs**
- **Find a model, such as Musa's, that fits your data**
- **Use to predict reliability or repair costs**

Software Reliability Modeling



Slideplayer.com



Department of Computer Science and Engineering
The Chinese University of Hong Kong

6

Part 4

Measuring the Structure of a Program

- **Introduction**
- **Some Popular Structure Metrics**
- **Cohesion and Coupling**
 - Coupling
 - Cohesion
- **Measures of Data Flow**

➤ Introduction

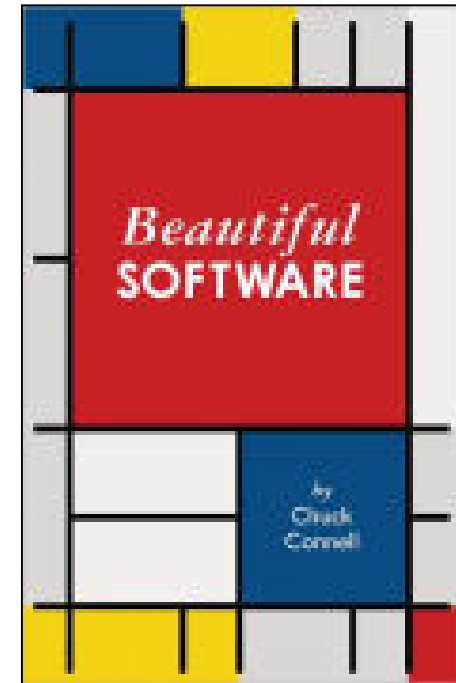
- Some Popular Structure Metrics
- Cohesion and Coupling
 - Coupling
 - Cohesion
- Measures of Data Flow



Possible Goals for Measuring Software Structure (1 of 5)

To identify and *admire the beauty* of its architecture?

That's really beautiful code!



Possible Goals for Measuring Software Structure (2 of 5)

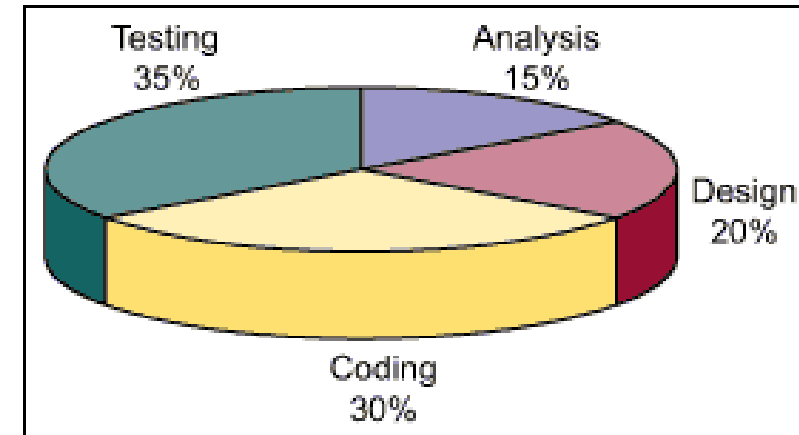
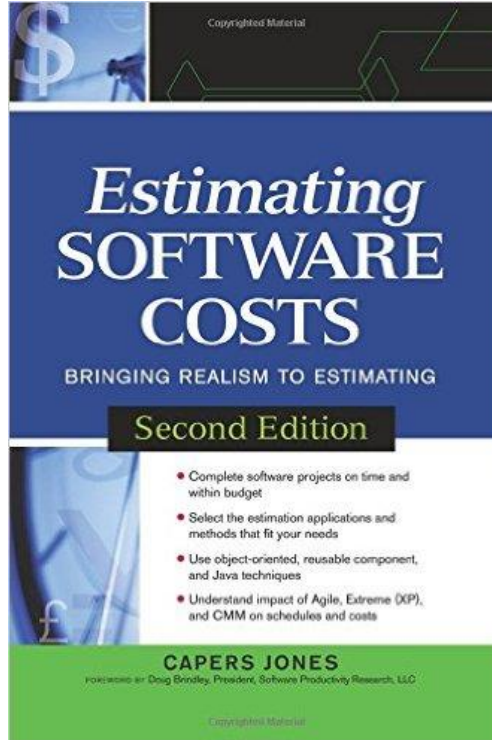
To help us identify *structural approaches* that are *more successful* than others

- Less error prone
- Easier to test
- Easier to understand
- Easier to maintain



Possible Goals for Measuring Software Structure (3 of 5)

To help us *estimate the effort* to produce the software



Possible Goals for Measuring Software Structure (4 of 5)

To help us estimate the *quality* of the software



Possible Goals for Measuring Software Structure (5 of 5)

To help us devise a *more effective test plan* for the software

Software Testing Abilities



Does Structure Relate to These Things?

Intuitively, *we believe that the structure of the software relates to its quality* and to the effort required to develop, test and support it:

- Ease of programming
- Ease of understanding
- Ease of testing and maintaining

So our *information needs* tend to be things like:

- What aspects of software structure can help *forecast* development effort and quality?
- *How should I test this software?*
- *How can I improve* my software structure?
- *How much has it improved?*

How Can We Measure Software Structure?

Many have attempted to devise ways of measuring structural aspects of software to see if they can show more specific relationships

- For example
 - Halstead's attempt to define programming difficulty and level of the language
 - McCabe's complexity measures (to be discussed later)

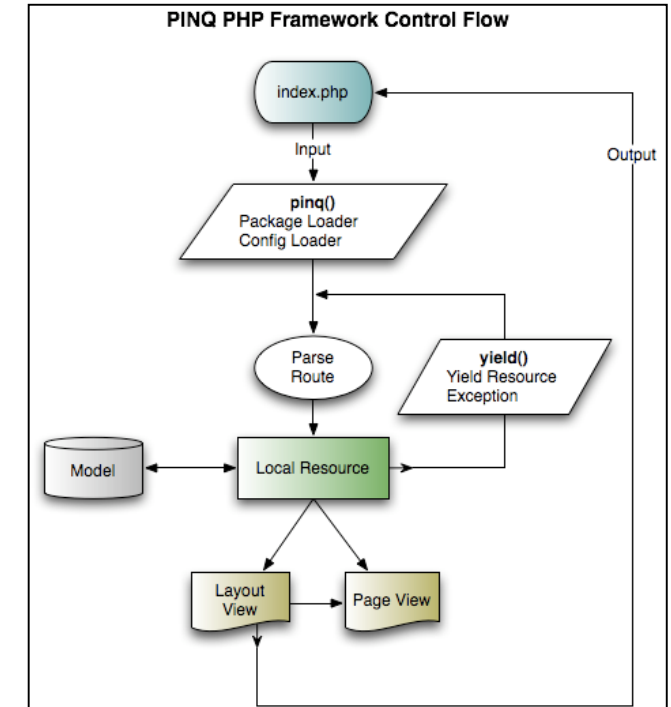
In this lecture we will explore *several of the most frequently cited and used methods of measuring software structure* and talk about their effectiveness

UTD There are Many Things We Could Measure

- Each ***Level of Abstraction*** has different Elements to Measure
 - Statement, function, method, class, package, sub-system, system
- We must understand how to ***describe*** each Element Measured
 - Syntax and semantics of the language or notation used to represent the software
- Eventually, we must measure ***attributes*** of ***individual components*** or ***elements*** of the design, code, requirements model, etc.

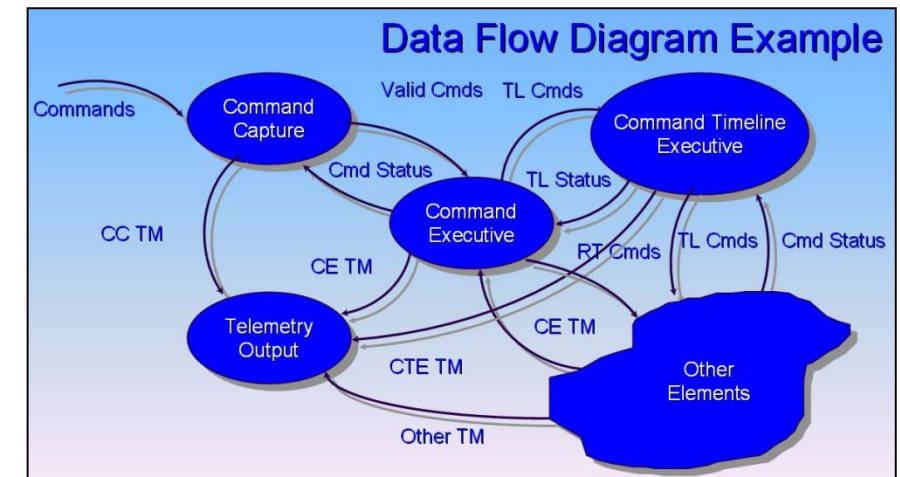
■ Control Flow

- The **sequence** in which things happen
 - Loops, parallelism, conditional execution, etc.

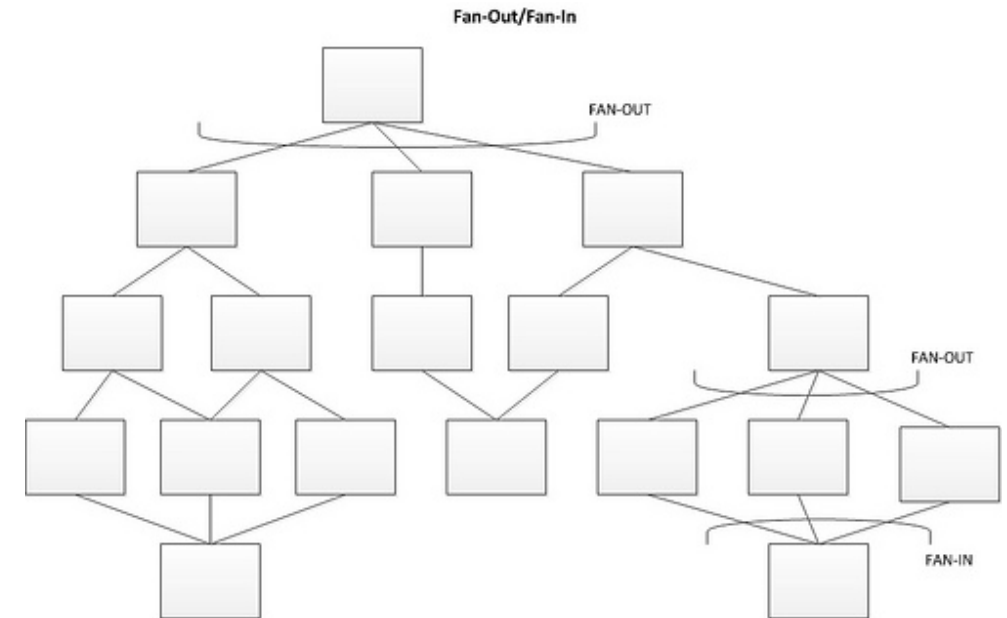


■ Data Flow

- The **trail of the data** as it flows through the program



- Introduction
- **Some Popular Structure Metrics**
- Cohesion and Coupling
 - Coupling
 - Cohesion
- Measures of Data Flow



Structural Fan-in and Fan-out

- ***Fan-in*** – the number of parent modules (the number of modules that call or utilize this module)
 - Goal: high fan-in at the lower levels of the hierarchy, such as procedures
- ***Fan-out*** – the number of subordinate modules (the number of modules that this module calls)
 - Goal: fan-out should be from 5 to 9 (7 ± 2).
 - This is based on studies of human psychology

- Introduction
- Some Popular Structure Metrics
- **Cohesion and Coupling**
 - **Coupling**
 - Cohesion
- Measures of Data Flow

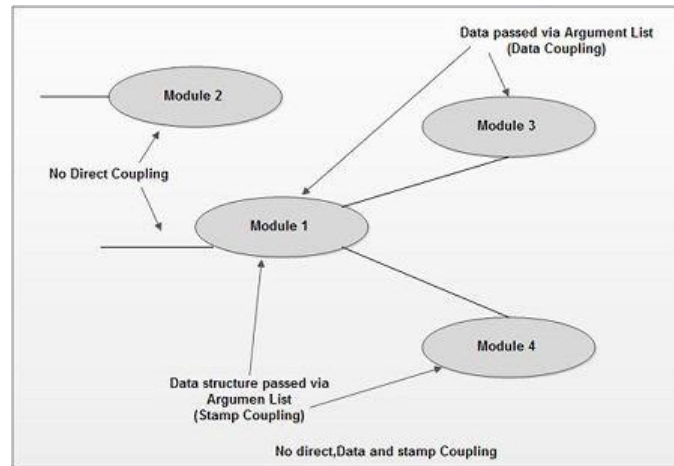
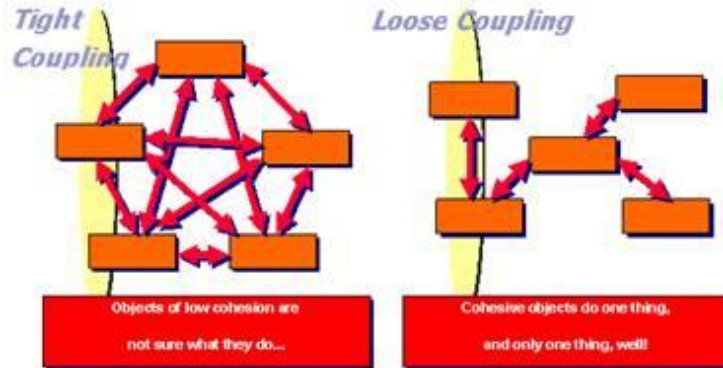


UTD Coupling of Modules or Methods -- Overview

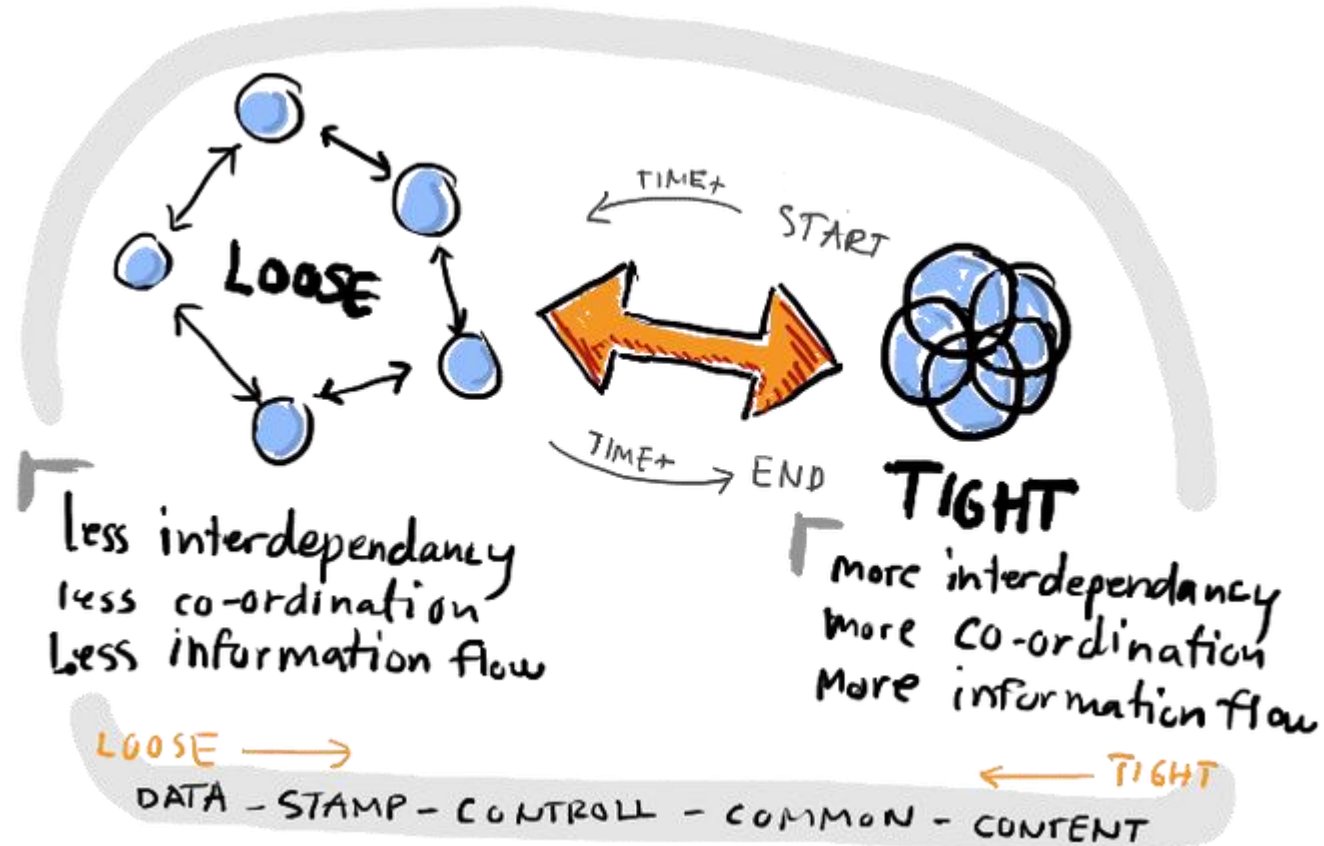
- **Coupling** refers to the *degree of independence* of a program module or method.
 - If measured for a **single module**, it indicates *the extent to which the module can function (or be understood)* without the use of other modules
 - If measured for a **pair of modules**, it indicates *the extent to which the two modules depend on each other*
- **Goal:**
 - Low or loose coupling is associated with *readability, testability* and *low maintenance cost*
 - Tight coupling is associated with *high maintenance cost*

The original concept was developed by Stevens, et. al. (see references)

Coupling Diagrams



Ecomputernotes.com



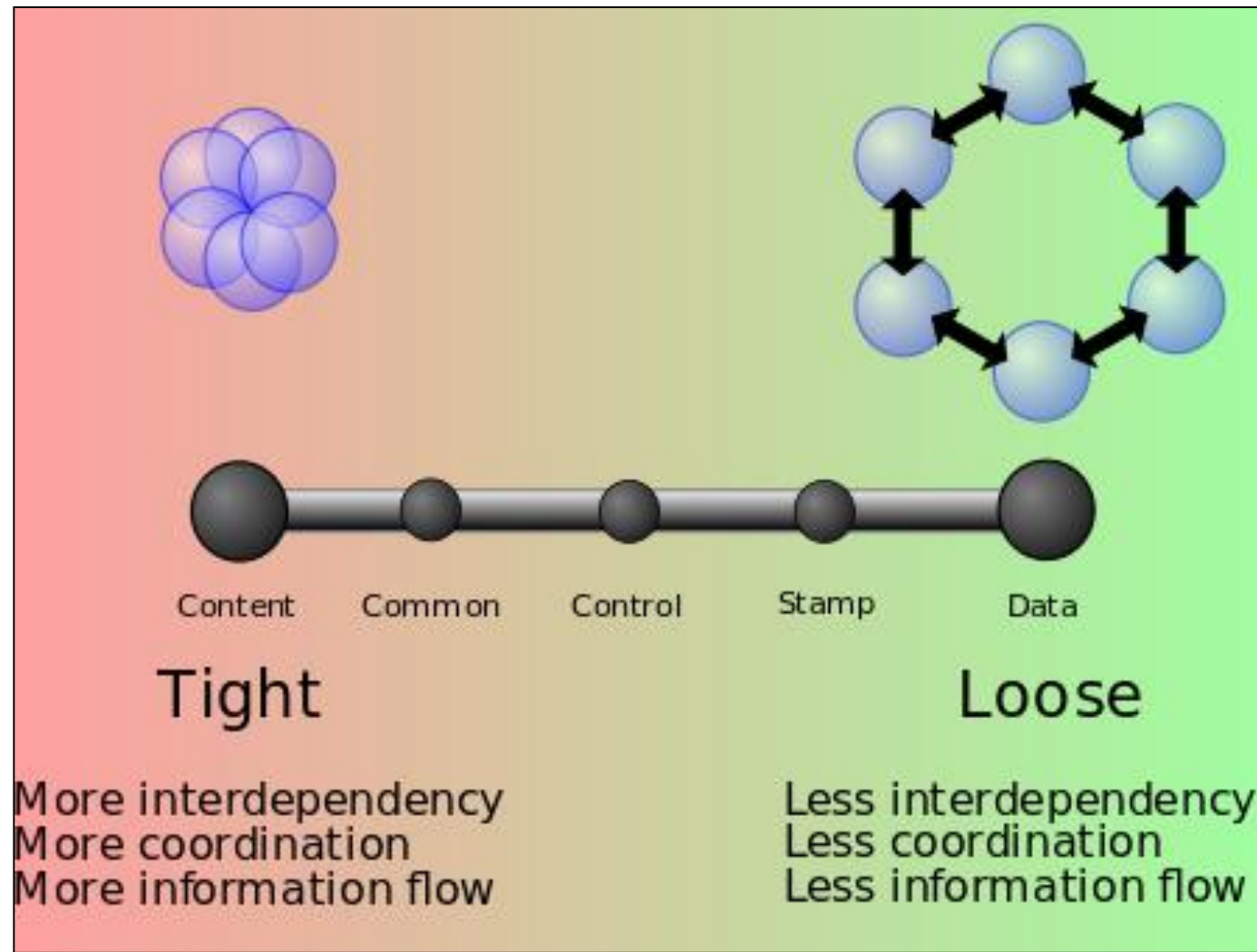
Dailyfintech.com

Coupling describes how much the modules or methods depend on other modules/methods.

Disadvantages of Tight Coupling

- Assembly / compiling of modules or methods may be ***more complicated*** and take more time, because you have to assemble / compile all of the affected ones
- ***Testing*** of a module or method ***is harder***
 - It involves interaction with other modules
- ***Changes*** to one module or method ***affect other modules or methods***
 - Consequences of a change are easily overlooked
 - It is ***harder to test the consequences*** of a change
- ***Reuse*** of a module is ***more error prone***
- Software ***performance*** may be ***affected due to*** the ***overhead*** of message/parameter passing and interpretation

UTD Degrees of Coupling (Myers' Classification)



En.Wikipedia.org

From tightest to loosest (Myers' classification)

- **Content coupling**: one module modifies or relies on the **internal data** or other workings of another module.
- **Common coupling**: two modules **share the same global data**. If the global data is changed in format or content, it affects all modules that use it.
- **External coupling**: two modules **share an externally imposed data format, communication protocol or device interface**.
 - For example, both use the same communication interface
- **Control coupling**: one module **controls the flow** of another, by passing it information on what to do (such as a flag).

From tightest to loosest (continued):

- **Stamp coupling (data structure coupling):** modules **share a composite data structure** and use different parts.
- **Data coupling:** modules share data through **parameters** or other means.
 - For example, a subroutine call
- **Message coupling:** modules communicate by **message passing**.
 - This is the loosest form of coupling
- **No coupling:** modules **do not communicate** with each other.

These degrees were proposed by Myers in Stevens, et. al. (see references)

Coupling Measures for OO Designs

- **There are three categories of coupling for a class**
 - ***Afferent coupling***: The number of responsibilities
 - ***Efferent coupling***: The number of dependencies
 - ***Total coupling***: **Afferent coupling + Efferent coupling**
- **And there's a measure computed from coupling info**
 - ***Instability***: The ratio of efferent coupling to total coupling
- **Other forms of coupling have been defined**
 - Different design methodologies may have different forms of coupling

See Fenton, section 9.4.1 for further examples and details

How to Measure Coupling

(slide 1 of 2)

There are *many proposed methods of measuring coupling*, varying somewhat with the kind of programming and other factors

For *procedural languages*, Stevens et. al. proposed the following formula for measuring the *coupling of a single module or method*:

- For data and control flow coupling:
 - d_i = number of input data parameters
 - c_i = number of input control parameters
 - d_o = number of output data parameters
 - c_o = number of output control parameters
- (continued on next slide)

How to Measure Coupling

(slide 2 of 2)

- For global coupling:
 - g_d = number of global variables used as data
 - g_c = number of global variables used as control
- For external coupling:
 - w = number of modules called (fan-out)
 - r = number of modules calling this one (fan-in)

Formula for coupling:

- $$C = 1 - \frac{1}{d_i + 2c_i + d_o + 2c_o + g_d + 2g_c + w + r}$$
- A large value indicates tight coupling
- Loose coupling -> .5-.7; tight coupling -> .9-1.0

UTD Fenton and Melton's Measure of Coupling¹ for a Pair of Modules or Methods

$$C(x,y) = i + n / (n+1)$$

- **x** and **y** are modules or methods
- **n** = the number of interconnections between x and y
- **i** = the level of the tightest coupling between x and y
 - **i** = **5** for content coupling
 - **i** = **4** for common coupling
 - **i** = **3** for control coupling
 - **i** = **2** for stamp coupling
 - **i** = **1** for data coupling
 - **i** = **0** for no coupling

¹ Fenton and Melton (see references)

Observations on Coupling

There are many studies that suggest *tight coupling* is associated with higher *cost*, *higher error rates*, and greater *difficulty in developing, testing and maintaining* software

However there are many variations on exactly how to measure coupling

This is an example of the kind of situation where the engineer finds something that works and uses it whereas the researcher spends countless hours trying to define a superior approach.

Advice: always use common sense. Don't put blind trust in any measurement that is as imprecise as this one.

Use of Coupling in Real Software Development

- As a principle of good design, programmers should *always seek to have loose coupling*.
- When coupling is needed, programmers should always *document what modules are affected by any messages, variables, parameters, etc.*
- *Measurements of coupling* can help you *identify areas of the code* that should be *redesigned or refactored* to make them simpler and less coupled, if possible.
- Identifying the degree and nature of coupling can help you *define more appropriate test approaches*.

- Introduction
- Some Popular Structure Metrics
 - ***Cohesion and Coupling***
 - Coupling
 - **Cohesion**
- Measures of Data Flow

Cohesion refers to *how well the parts of the module* or the *methods of a class* relate to each other.

- A module/class is cohesive if all of its functions are closely related to each other.

Cohesion is desirable because it is *easier to understand* the module or class if there is a *single, unifying theme* for what it does.

- The module or class makes sense as a meaningful unit
- Various functions related to that theme would tend to use the same terminology, have the same sorts of exceptions, the same data types, and the same sorts of errors.

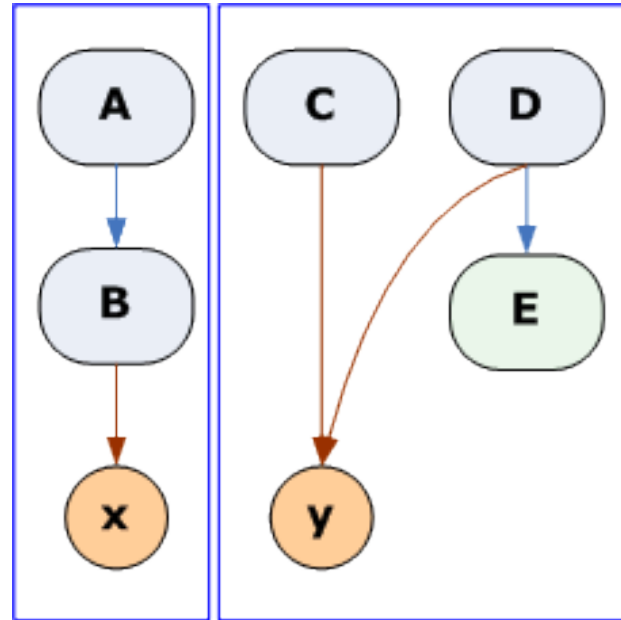
Most work on cohesion has been focused on object oriented methods rather than procedural methods.

Evaluating Cohesion

A simple way to judge cohesion is to determine how succinctly the module can be described.

- A ***short and precise*** sentence generally describes a **cohesive** module.
 - ***"This module handles input/output"***
 - The above module could easily have a descriptive name, such as ***"InputOutputFunctions"***
- A ***longer and less precise*** sentence suggests a **non-cohesive module**.
 - ***"This module factors the data, performs various services and handles the I/O"***
 - Words such as "various" and "assorted" and "variety" in the description are typical of non-cohesive modules

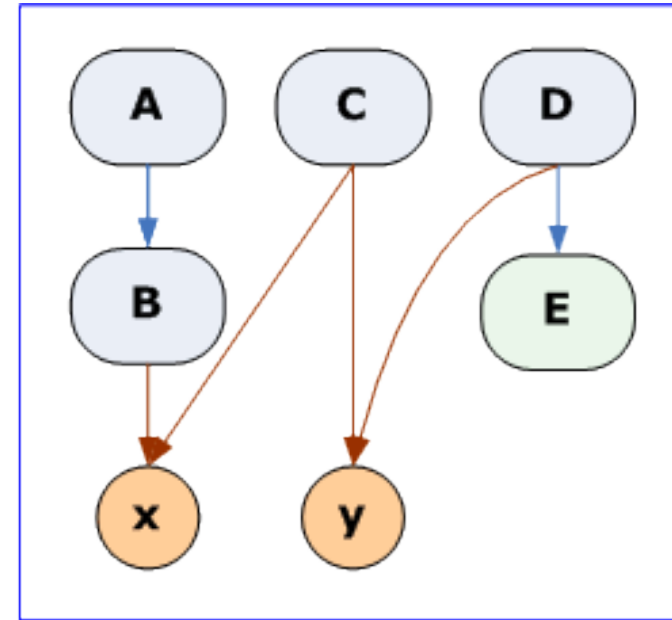
Example



LCOM4 = 2

LOW COHESION

This module or class has two separate functions or groups of methods that are unrelated.



LCOM4 = 1

HIGHER COHESION

This module or class has functions or methods that are strongly related

Advantages of Cohesion

- ***If you need to replace*** part of the module or class it is likely that you will replace most or all of it
 - So you can simply replace the whole thing rather than removing parts of it and leaving other parts alone
 - You avoid inadvertently damaging one part while modifying another part
- ***Interfaces*** to cohesive modules tend to be ***cleaner and more coherent***
- **High cohesion promotes *encapsulation***
 - Placing of related data and functions into a single component
- ***Low cohesion*** generally means inappropriate design with ***high complexity***

Observations About Cohesion

Generally speaking, *high cohesion and low coupling tend to go together*

- But not always

Cohesion of a class may mean that *the methods in that class are strongly coupled*, which may make it *harder to test and maintain*

- This is a potential drawback, which illustrates why one should not always insist on high cohesion

Cohesion of a procedure has similar drawbacks regarding its internal components.

Various metrics have been proposed, under the name “Lack of cohesion metric” or LCOM. Most make more sense for OO software.

LCOM1: This is measured for two methods in a class or two separate sub-functions of a procedure

Let **P** = The **number of disjoint sets of variables** accessed by the two methods/sub-functions

Let **Q = 1** if the two methods/sub-functions access at least one common variable; otherwise, **Q = 0**

$$\mathbf{LCOM1 = P - Q}$$

If **P = 0**, the methods/sub-functions are cohesive

If **P > 0** they are not, and could be separated.

Drawbacks of LCOM1

- **Makes more sense for pairs of methods than for classes**
- **Only one value ($LCOM1 = 0$) is defined for cohesive situations, which means it doesn't measure the "degree of cohesiveness"**
- **The definition doesn't account for certain classes of global and other shared variables (details vary with the specific OO methodology being used)**

For more on LCOM1 see Chidamber and Kemerer in reference list.

LCOM2 and LCOM3

Both measure the degree of cohesion

m = the number of methods in a class or sub-procedures in a procedure

a = the number of variables or attributes in a class or sub-procedure

mA = the number of methods that access a specific variable or attribute
A

Sum(mA) = sum of the mA values for all attributes or variables

LCOM2 = $1 - \text{Sum(mA)} / (m * a)$

- This ranges from 0 to 2
- 0 is good, 1 is not good, 2 is very bad

LCOM3 = $(m - \text{Sum(mA)} / a) / (m - 1)$

- This ranges from 0 to 1
- 0 is good, 1 is not good

see Henderson-Sellers
in reference list.

This measures the number of connected components in a class.

- A connected component is **a set of related methods** (and class level variables). Ideally there should be only one connected component in a class.
- **Two methods are related if:**
 - They both access the same class-level variable, or
 - One of them calls the other one
- **To measure LCOM4, you determine which methods are related and draw a directed graph, showing the relationships**

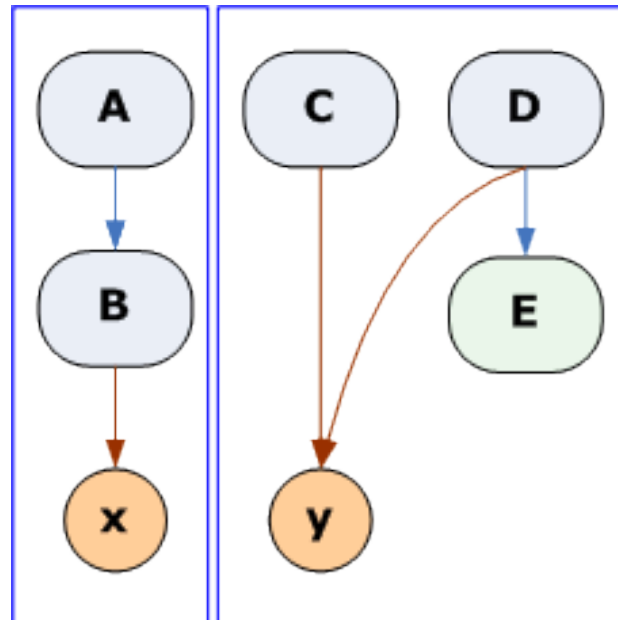
We will discuss directed graphs later.

Interpreting LCOM4

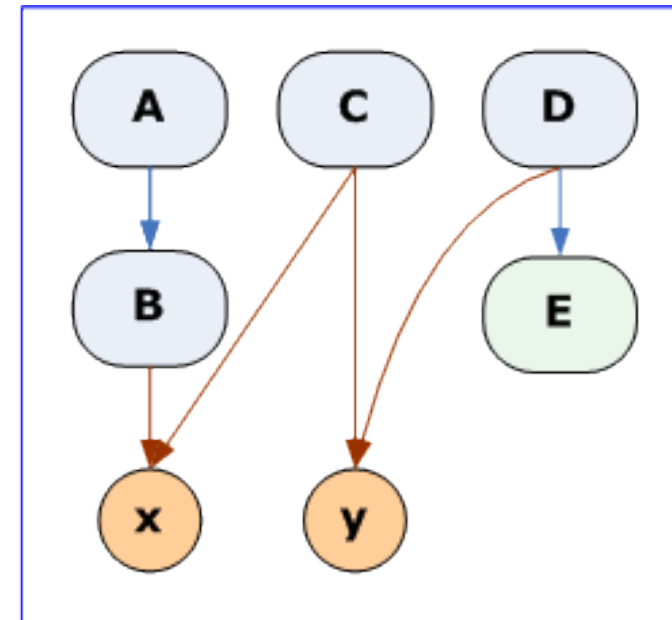
LCOM4 = 1 means high cohesion

LCOM4 > 1 means lower cohesion

LCOM4 = 0 means a class with no methods



LCOM4 = 2



LCOM4 = 1

Summing Up Cohesion

- **There are many measures of cohesion, including some we have not discussed**
 - but none have been universally accepted
- **Cohesion is a good design goal for most software**
- **Good cohesion often means low coupling, which is also a good design goal**
- **But measures of cohesion vary in their usefulness, depending on the design methodology and other factors**
 - So use them with caution
- **Lack of cohesion generally means that testing will be more difficult**

- Introduction
- Some Popular Structure Metrics
- Cohesion and Coupling
 - Coupling
 - Cohesion
- **Measures of Data Flow**

A Fundamental Issue with Data Flow Measures

IEEE Standard 982.2 defines a series of *informational flow complexity measures*

But the software development community has adopted a number of *variations* on these

- None of these have been accepted as a standard, because there are so many variations.
- Many of these *depend on the specific methodology or language* being used

We will mention some of the differences

Informational Fan-in (IFIN): Information flow into a procedure or method

$$\text{IFIN} = \text{PC} + \text{PR} + \text{GVR}$$

PC = number of procedures calling this one

PR = number of parameters read

GVR = number of global variables read

Informational Fan-in (IFOUT): Information flow out of a procedure

$$\text{IFOUT} = \text{CP} + \text{PW} + \text{GVW}$$

CP = number of procedures that this one calls

PW = number of parameters written to [by reference]

GVW = number of global variables written to

There are several variations on exactly what is counted as information flow into or out of a procedure or method

For more info, see Henry and Kafura in reference list

Informational Fan-in x Fan-Out (IFIO):

$$\text{IFIO} = \text{IFIN} * \text{IFOUT}$$

- This is supposedly a good measure of the effort required for implementing the procedure
- But it is not necessarily a good measure of its overall complexity
 - i.e., IFIO is not necessarily a good measure of how hard it is to understand, test or maintain)

Additional Measures of Data Flow - IFC

Informational Flow Complexity (IFC):

$$\text{IFC} = \text{IFIO} * \text{IFIO} \quad [\text{in other words, } \text{IFIO}^2]$$

- This is the IEEE standard definition

$$\text{IC1} = \text{LOC} * \text{IFIO} \quad [\text{in other words, } \text{size} * \text{IFIO}]$$

- This is a widely used definition of information flow complexity

- **Some authors believe that IFC is a good measure of how hard it is to understand, test or maintain the software**

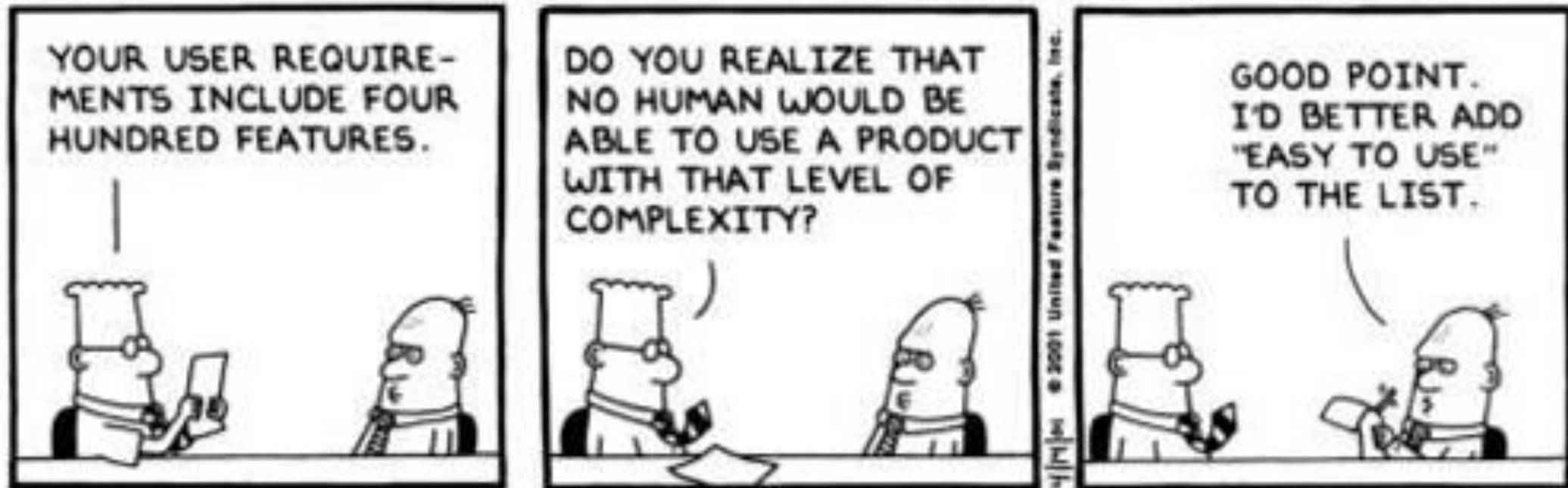
Regardless of how measured, ***high information flow complexity is generally not a good thing*** with most modules or methods

- Procedures or methods with high information flow complexity are good candidates for redesign because they may be hard to understand
 - They may require extensive testing
- On the other hand, ***some software*** may be designed to ***intentionally*** have a ***method or module responsible for a large amount of information flow***

Things You Can Discover from Information Flow Metrics

- **More than one function is required to implement a procedure** (*lots of information flow between the functions*)
 - Is there a good reason for this from a design perspective, or is this a candidate for redesign?
- ***Stress points* in a system** (places where there is a lot of information traffic)
 - This may indicate parts of the software that will have performance issues
- **Excessive *functional complexity***
 - The difficulty of implementing and testing a function due to the complexity of what it must accomplish

- This is a current research topic
- See Lavazza and/or Abran in the reference list



END OF Part 4

Part 5

Measuring Software Complexity

Contents

- **Complexity: what and how to measure**
- **Structured Programs and Flowgraph Analysis**
- **Measures of Complexity**
- **Closing Remarks**

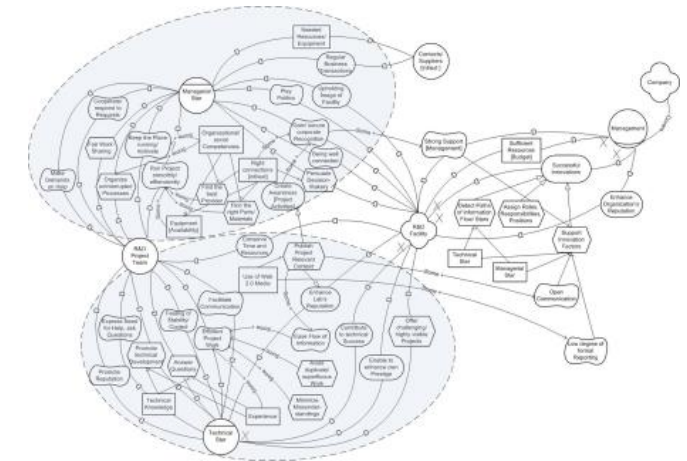
- ***Complexity: what and how to measure***
 - **Structured Programs and Flowgraph Analysis**
 - **Measures of Complexity**
 - **Closing Remarks**

We tend to think that complex software is more difficult to develop, test and maintain and has greater quality problems.

But what do we mean by complexity?

Dictionary definitions of complex:

1. Composed of ***many interconnected parts***
2. Characterized by a very ***complicated arrangement of parts***
3. So ***complicated or intricate*** as to be ***hard to understand***



Complex vs Complicated

Complicated: being difficult to understand but with time and effort, **ultimately knowable**

Complex: having many interactions between a large number of component entities.

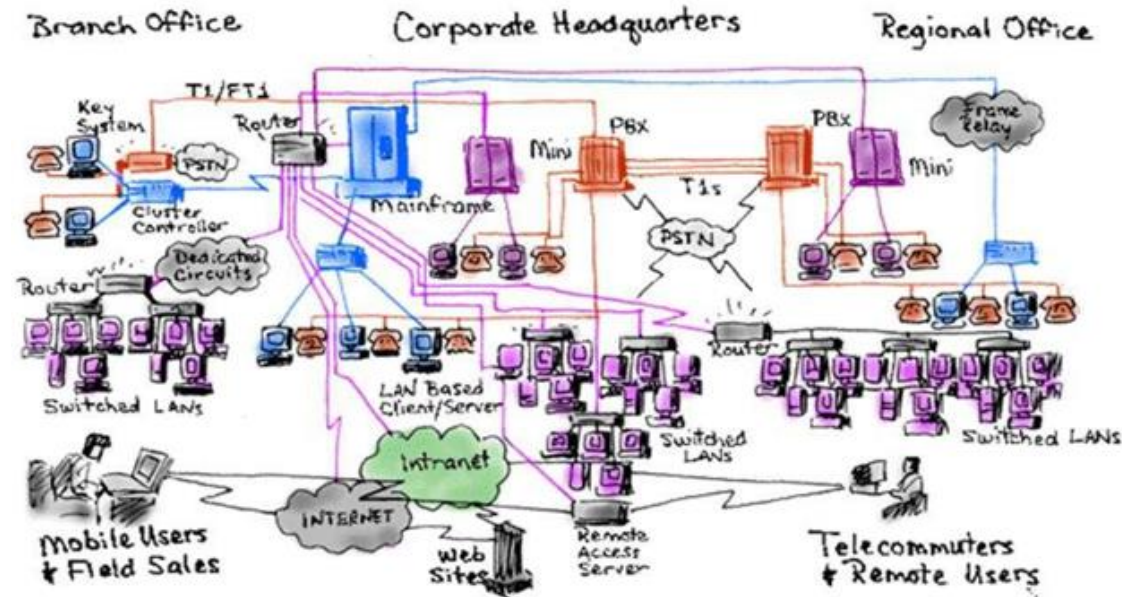
- As the number of entities increases, *the number of interactions* between them will **increase exponentially**
- It can get to a point where it would be **impossible to know and understand** all of them.



Hotel-r.net

Changing Complex Software

- Higher levels of complexity in software increase the risk of unintentionally interfering with interactions and so **increase the chance of introducing defects** when making changes.



Labs.Sogeti.com

- In more extreme cases, complexity can make **modifying the software virtually impossible**. Changes introduce more problems than they fix. This is called **inherent instability**.

Can We Measure Complexity?

Measures of complexity would need to address:

- the *parts* of the software,
- the *interconnections* between the parts,
- and the *interactions* between the parts.

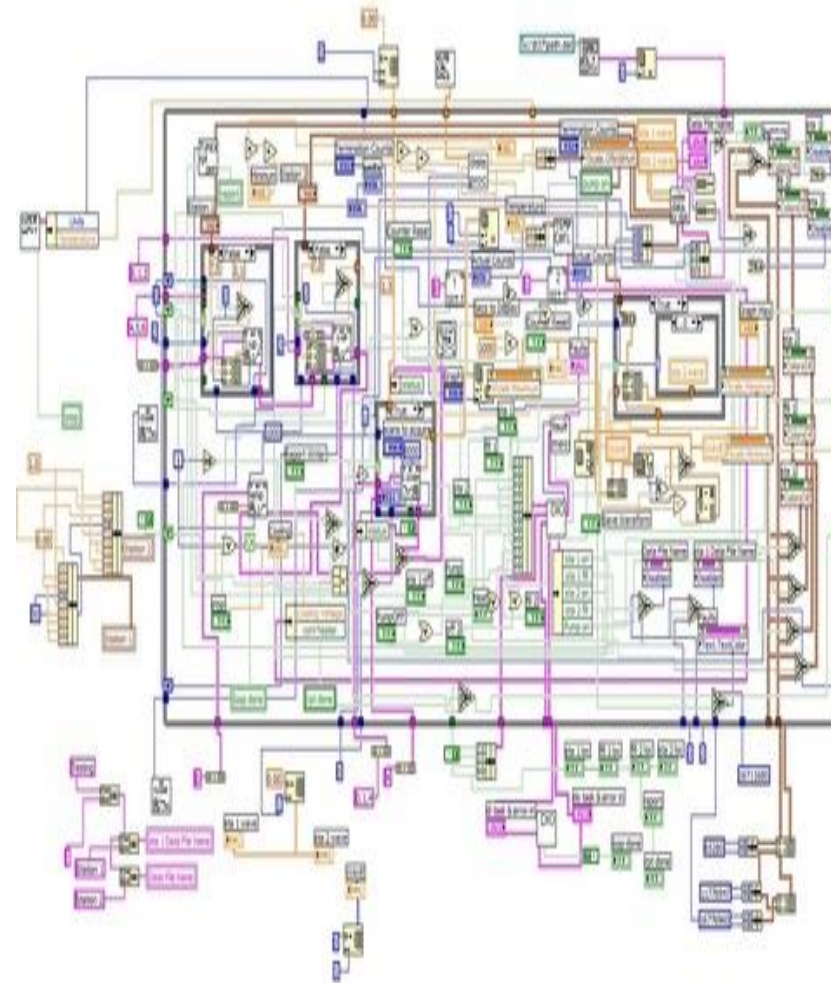
Information Need

- Something that will help us *estimate*
 - difficulty of programming,
 - difficulty of testing and maintaining,
 - expected level of quality
- Something that will *help us evaluate and improve our software* with regard to the above characteristics

How Can We Measure Complexity?

The ***base measures*** would quantify the attributes of:

- The ***parts*** or ***components*** of the software
- ***How many*** parts or components there are
- The ***arrangement*** of the parts
- The ***interactions*** of the parts



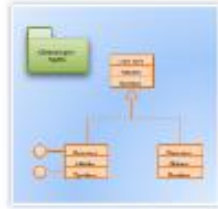
Combining the base measures into calculations that help us address our information needs, answering questions such as:

- What aspects of software structure can help **forecast** development effort and quality?
- ***Is my software structure good?***
- ***How should I test my software?***
- ***How can I improve*** my software structure?
- ***How much has it improved?***

We might learn something about the structure and complexity of software by measuring:

- ***Requirements***
 - Models, use cases, test cases
- ***Architecture and Design***
 - Models, design patterns, structure, control flow, data flow
- The **code** itself
 - Statements, variables, nesting, control flow, data flow
- The ***way the code is assembled*** to produce the final product
 - Load files, use of libraries

One Problem Is That There are Many Systems for Describing Software Structure



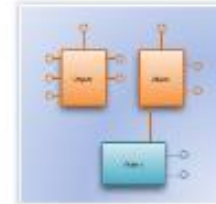
UML Model Diagram



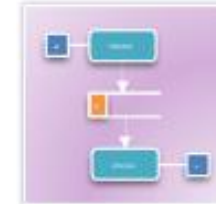
Windows 7 UI



Booch OOD



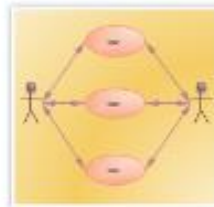
COM and OLE



Data Flow Model Diagram



Enterprise Application



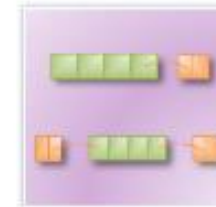
Jacobson Use Case



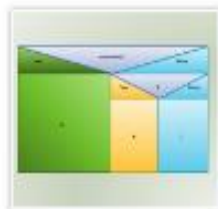
Jackson



Program Flowchart



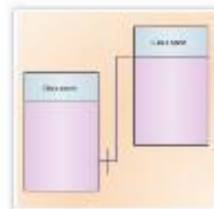
Program Structure



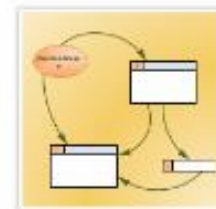
Nassi-Shneiderman



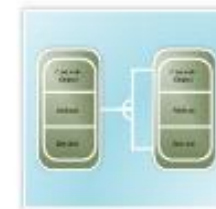
ROOM



Shlaer-Mellor OOA



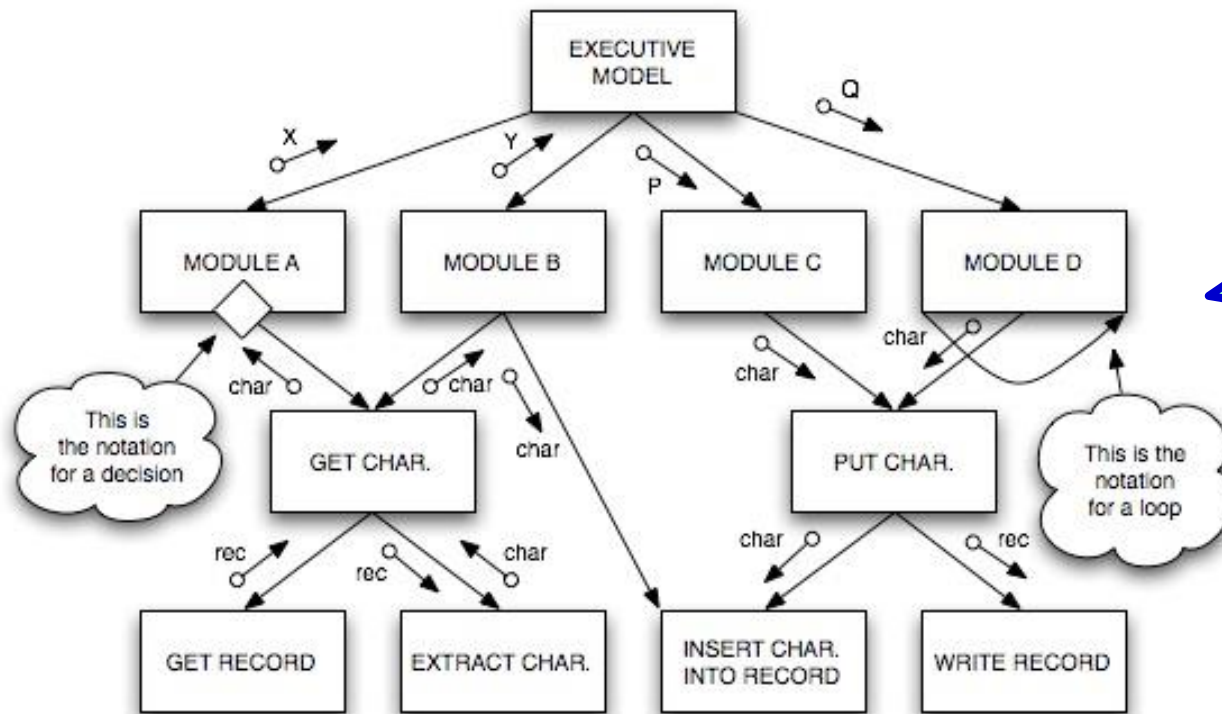
SSADM



Yourdon and Coad

Generally Speaking We Measure Complexity of Systems and of Components that Make up Systems

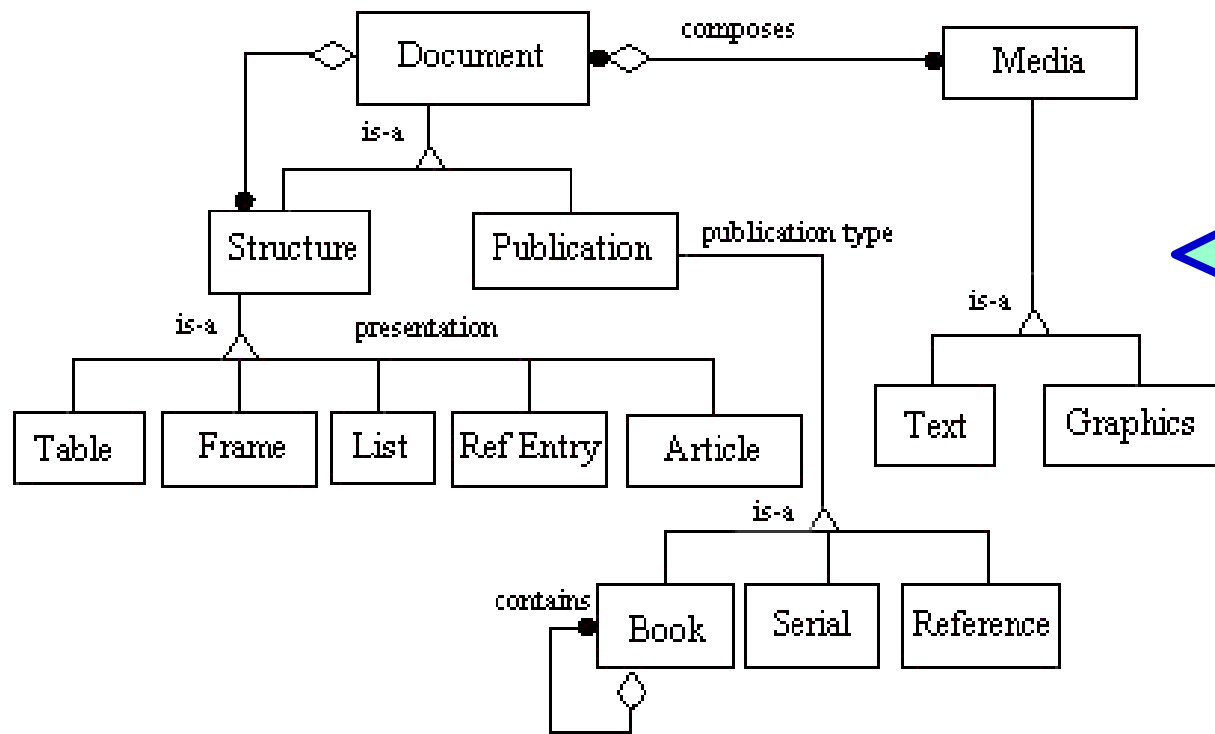
We usually start with the *architecture* of the system



This is the architecture of a system defined using *structured analysis*. There are complexity measures for the system and for the individual components.

With Object Oriented Systems, the Nature of the Components Varies with the Methodology

This means we must sometimes devise *methodology-specific measures*



This is the architecture of a system defined using *object oriented methodology*. There are complexity measures for the system and for the individual components.

Figure 1. Multimedia Document Model - Object diagram

Order of Presentation

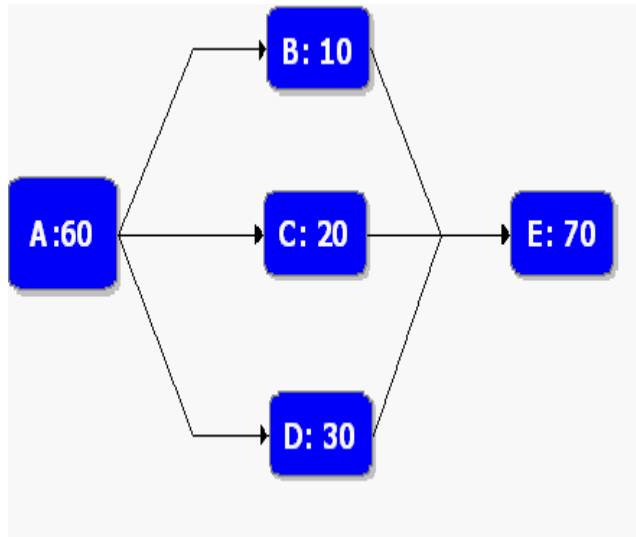
We will focus on complexity of *structured, procedural software*

- Because this is where most of the research has been focused
- Because the results apply to software in many different languages
- Because ***most of the results also apply to object oriented software***

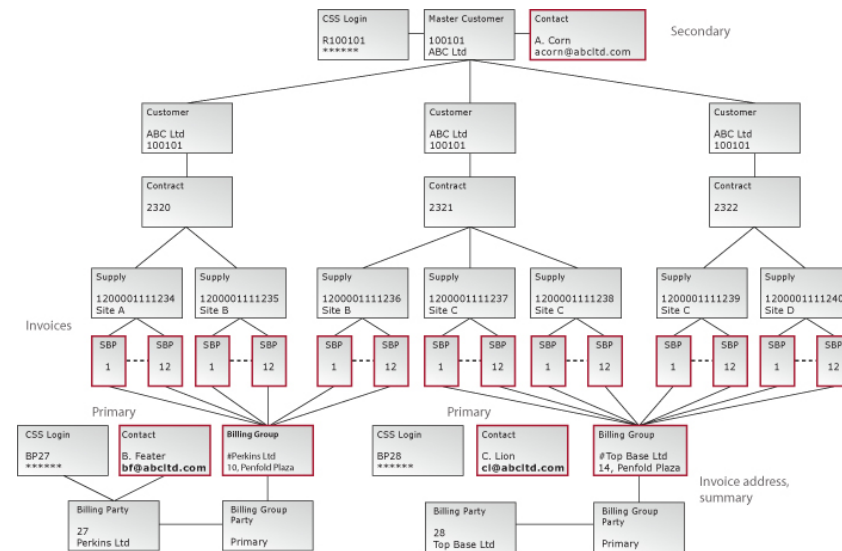
From time to time we will mention how the concepts are applied to *object oriented software*

System Level Complexity

Fundamentally, the *complexity* of a system depends on the *number of components* and the *number of links* between the components of the system



VS

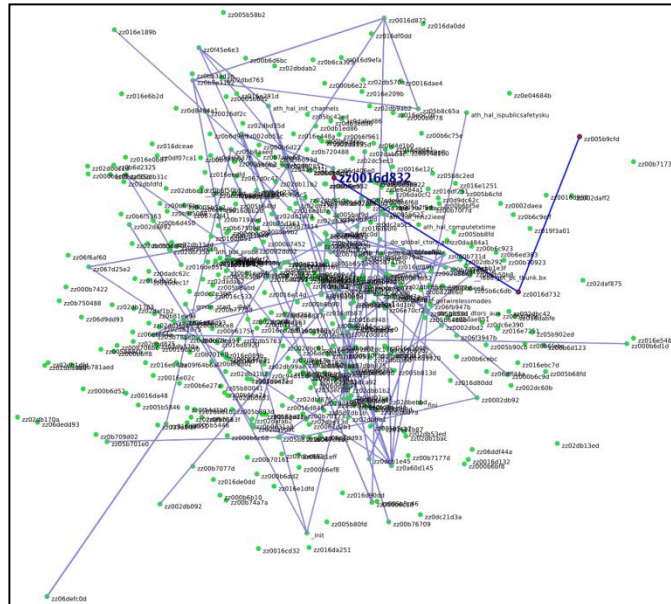


It can be ***further complicated*** by the degree to which the ***components share common elements*** (coupling)

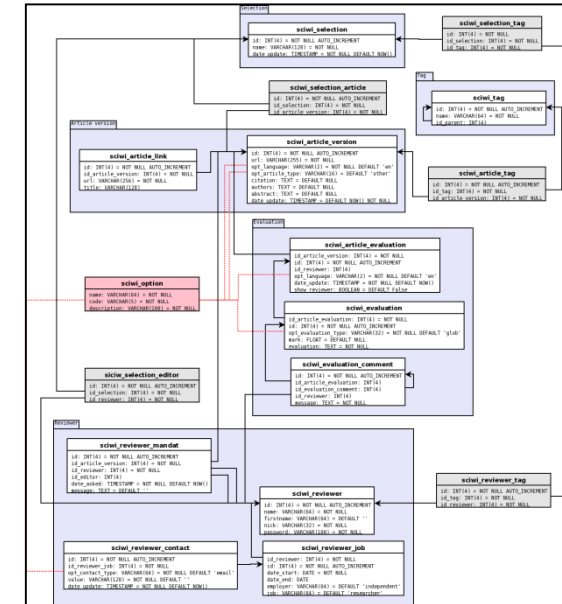
- Complexity: what and how to measure
- ***Structured Programs and Flowgraph Analysis***
- Measures of Complexity
- Closing Remarks

Control Flow Captures Major Complexity-related Attributes

Our intuitive notions of complexity would say that when there are *more parts* and more *complex ways they interact*, we have more complex software.

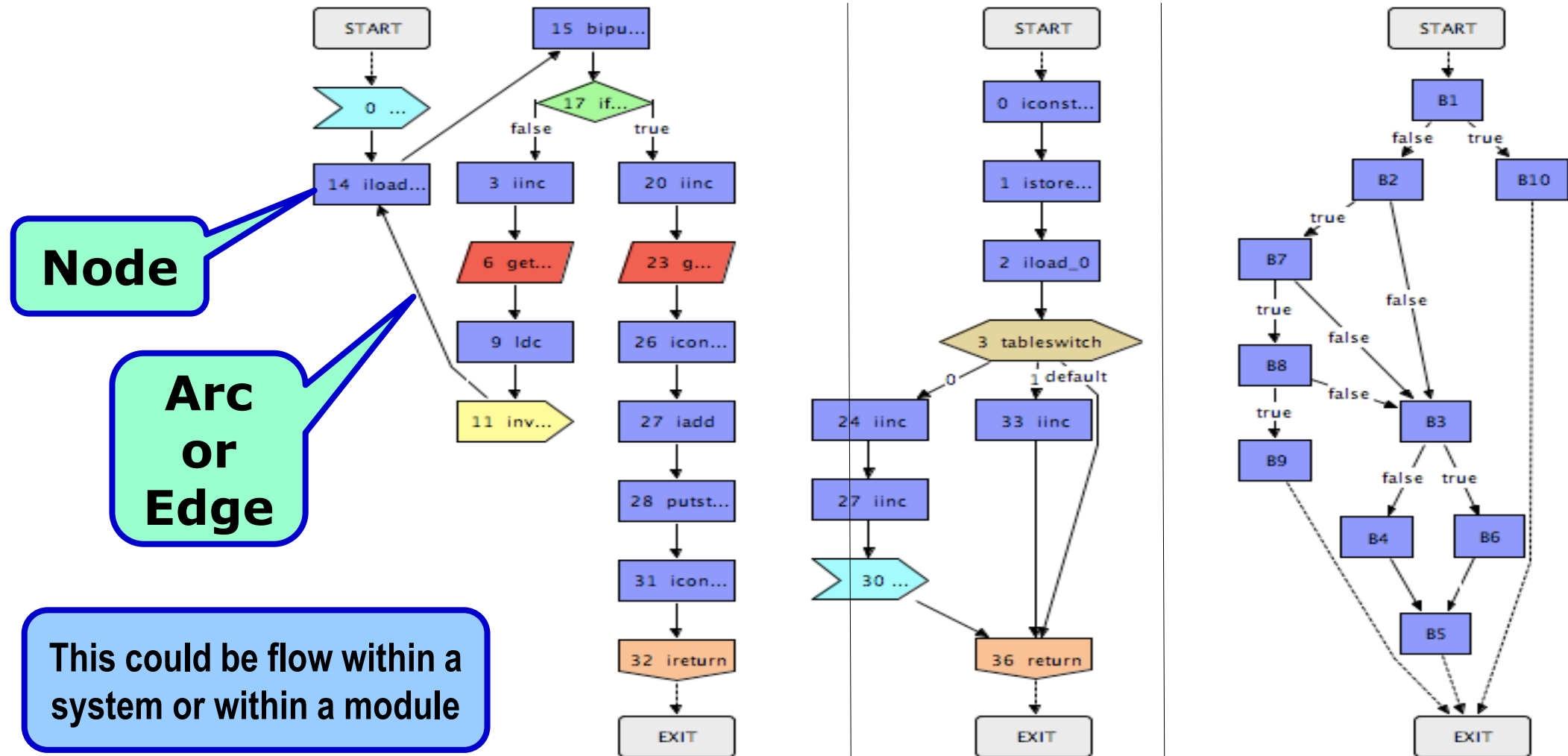


VS



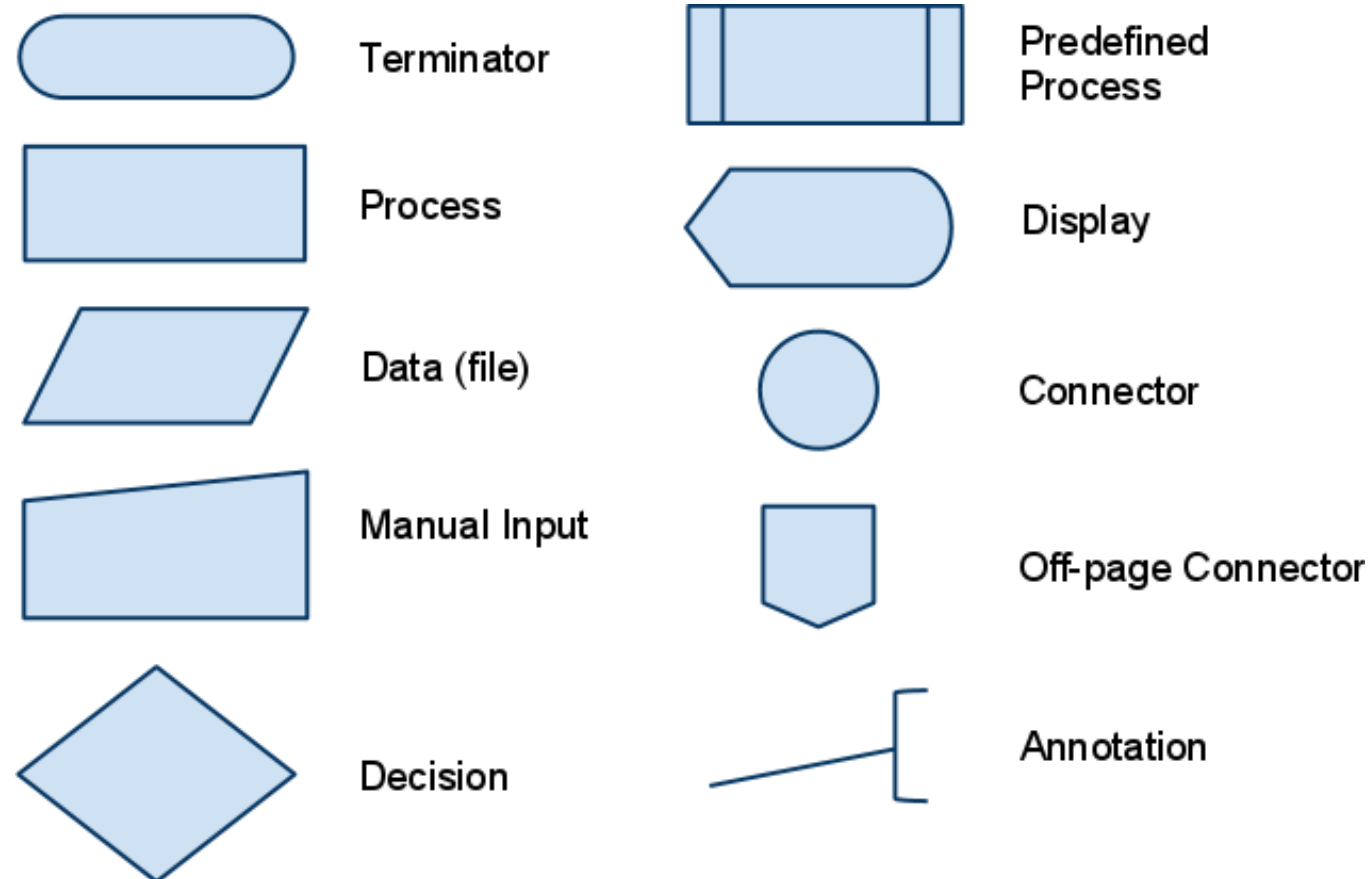
Many measures of complexity make use of control flow analysis.

Control Flow is Often Modeled with Directed Graphs



In Many Notations, the Shape of the Node Conveys the Nature of What it Represents

For example, flowcharts:



Notation To Be Used Here

(in these slides)

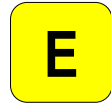
- **Arc or Edge**



A path between nodes

- **Procedure Node**

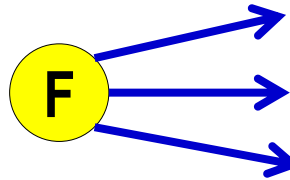
- A block of code. Any decisions are internal to the block. One exit.



Squarish shape,
Exactly one arc leaving

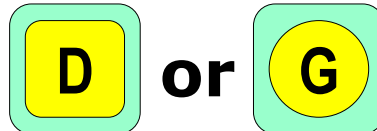
- **Predicate Node**

- One that makes a decision.

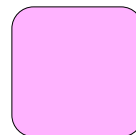


Round shape, Two or
more arcs leaving

- **Start Node**



- **Stop Node**



Colors of procedure and predicate nodes are not part of the notation.
Colors are used only to clarify points being made on a slide.

A flowgraph is a directed graph with

- ***One start node***, and
 - ***One end node***,
- **that has the following property:**
- ***Every other node lies on a path between the start node and the end node***

Notes:

- This notation works for any procedural programming language
- But not all languages can represent all possible flowgraphs
- Certain common language constructs have readily recognized flowgraph forms

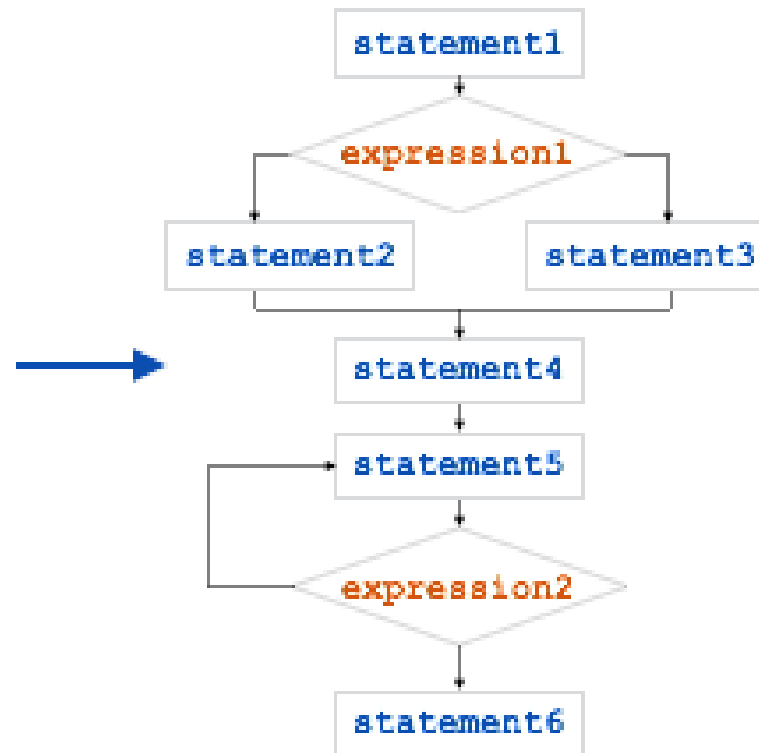
See later slides or Fenton,
page 379 for some examples.

Example: Code, Flowchart, and Flowgraph

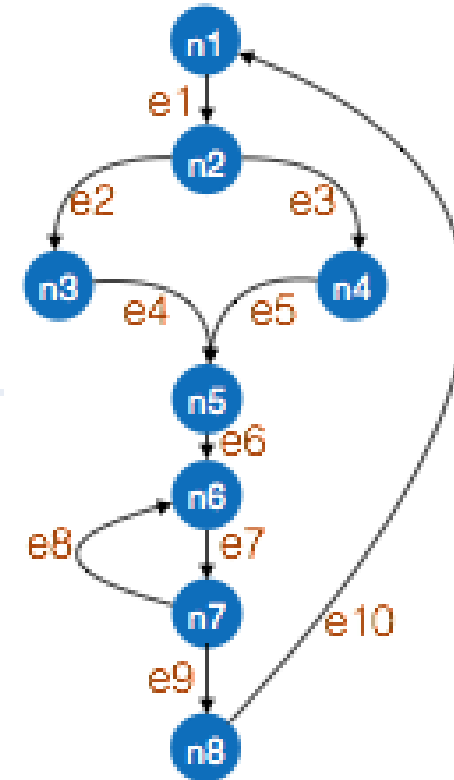
Code

```
statement1
If expression1
    statement2
else
    statement3
statement4
do
    statement5
while expression2
statement6
```

Flow-Chart



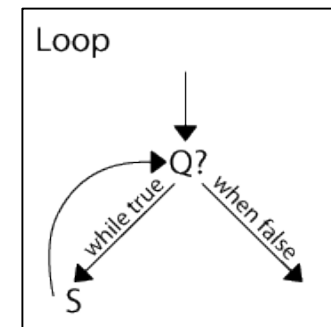
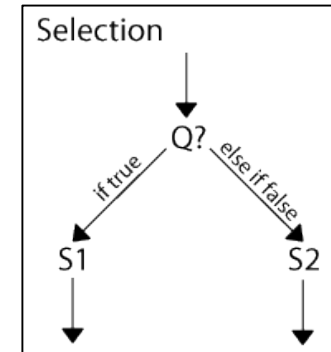
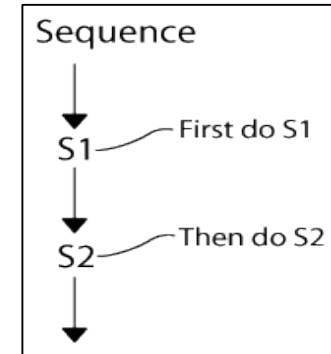
Flow-Graph



What is a Structured Program?

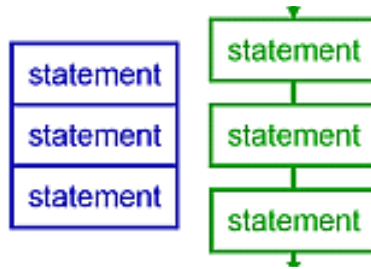
A structured program is one *constructed out of three fundamental control structures*:

- **Sequence** (ordered statements and/or subroutines)
 - Examples: $A = B + C$; $D = \text{FUNC}(E, F)$
- **Selection** (one or more statements is executed, depending on the state of the system)
 - Example: **If** C1 **Then** <true option> **Else** <false option>
- **Iteration** [loop] (a statement or block is executed until the program has reached a certain state)
 - Examples: **While**; **Repeat**; **For**; **Do... Until**

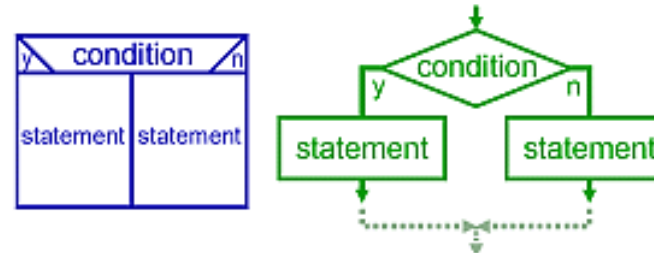


Structured Program Notation

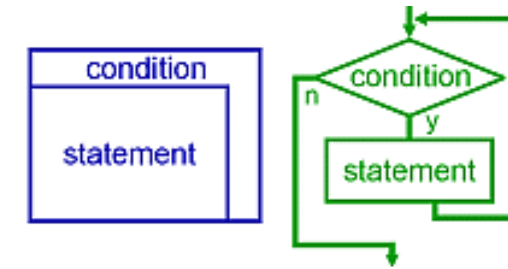
Sequence



Selection

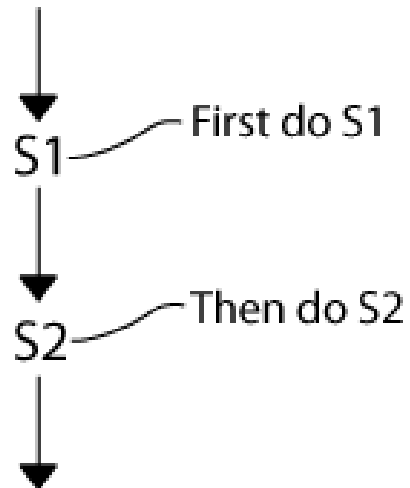


Iteration (Loop)

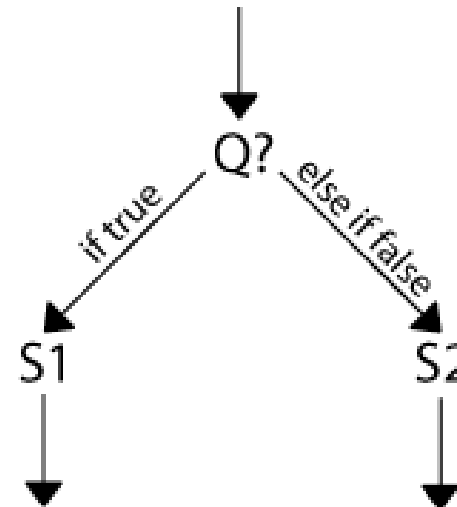


Blue: NS Diagram notation; **Green:** Flowchart notation

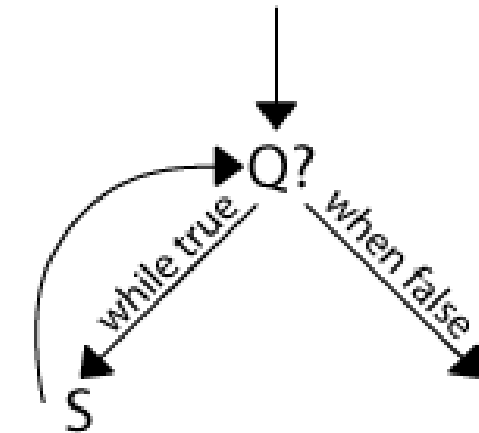
Sequence



Selection



Loop



These Three are Sufficient to Represent Any Program

The *structured program theorem*, also known as the Böhm-Jacopini theorem, says that the class of flowgraphs representing *the three control structures above can compute any computable function*

- **Note:** This does not necessarily mean it is the only way or the best way.
- The theorem simply states that it is **possible** to represent any function with only the three control structures.

Why Are Structured Programs Important?

Studies have shown that limiting the software to a small number of well defined control structures has these benefits:

- Easier to understand
- Less error prone
- Easier to analyze and test
- Easier to measure

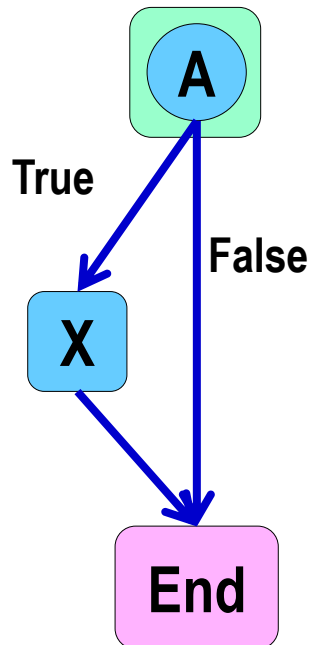
This started out as a theoretical concept, developed by Edsger Dijkstra and others.

It became more widely known when Dijkstra wrote his famous “*Go To Considered Harmful*”¹ letter to the editor of *Communications of the ACM* (in 1968).

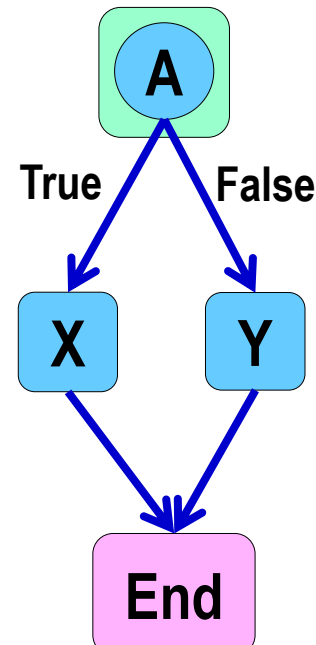
¹ See References

There May Be More Than One Flowgraph Representing A Particular Kind of Control Structure

Example: Two flowgraphs for selection



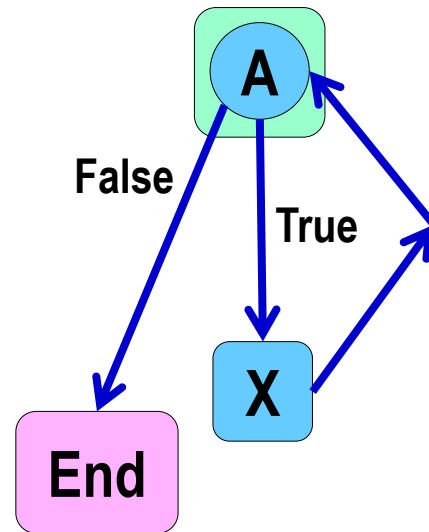
If A then X
(D₀)



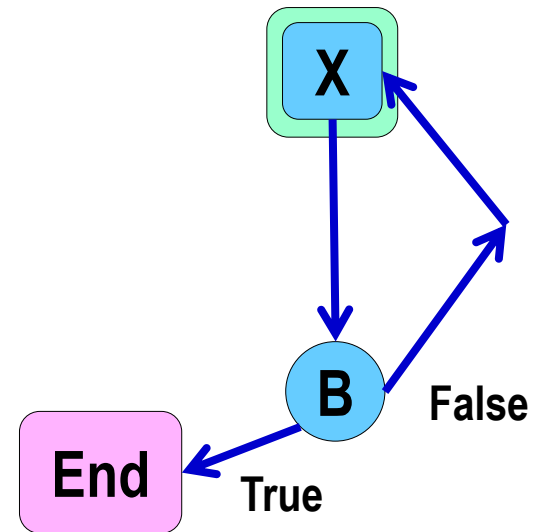
If A then X else Y
(D₁)

Each of these is also a "***prime***" flowgraph, meaning it cannot be reduced to a simpler form. We'll discuss this further in later slides.

Two Prime Flowgraphs for Iteration



While A Do
X
(D₂)



Repeat X
Until B
(D₃)

Prime Flowgraphs and D Notation

- A ***prime flowgraph*** is one that ***cannot be reduced*** (to a simpler flowgraph).
 - D_0 , D_1 , D_2 and D_3 are all prime.
 - See discussion of “reduction” in later slides.
- The D notation is a widely recognized way of denoting certain standard, prime flowgraphs.

If A then B
(D_0)

This is a standard type of flowgraph, known as a D_0 structured flowgraph.

The Flowgraphs D_0 - D_3 (and sequencing) Can Be Used To Represent Any Program

As a result, *some define a program to be "structured" only if it is represented by a combination of these flowgraphs.*

However, there are several additional prime flowgraphs that represent *commonly used language constructs* and that can greatly simplify some programs.

So different organizations and researchers have defined *additional prime flowgraphs* that may be permitted in "structured" programs.

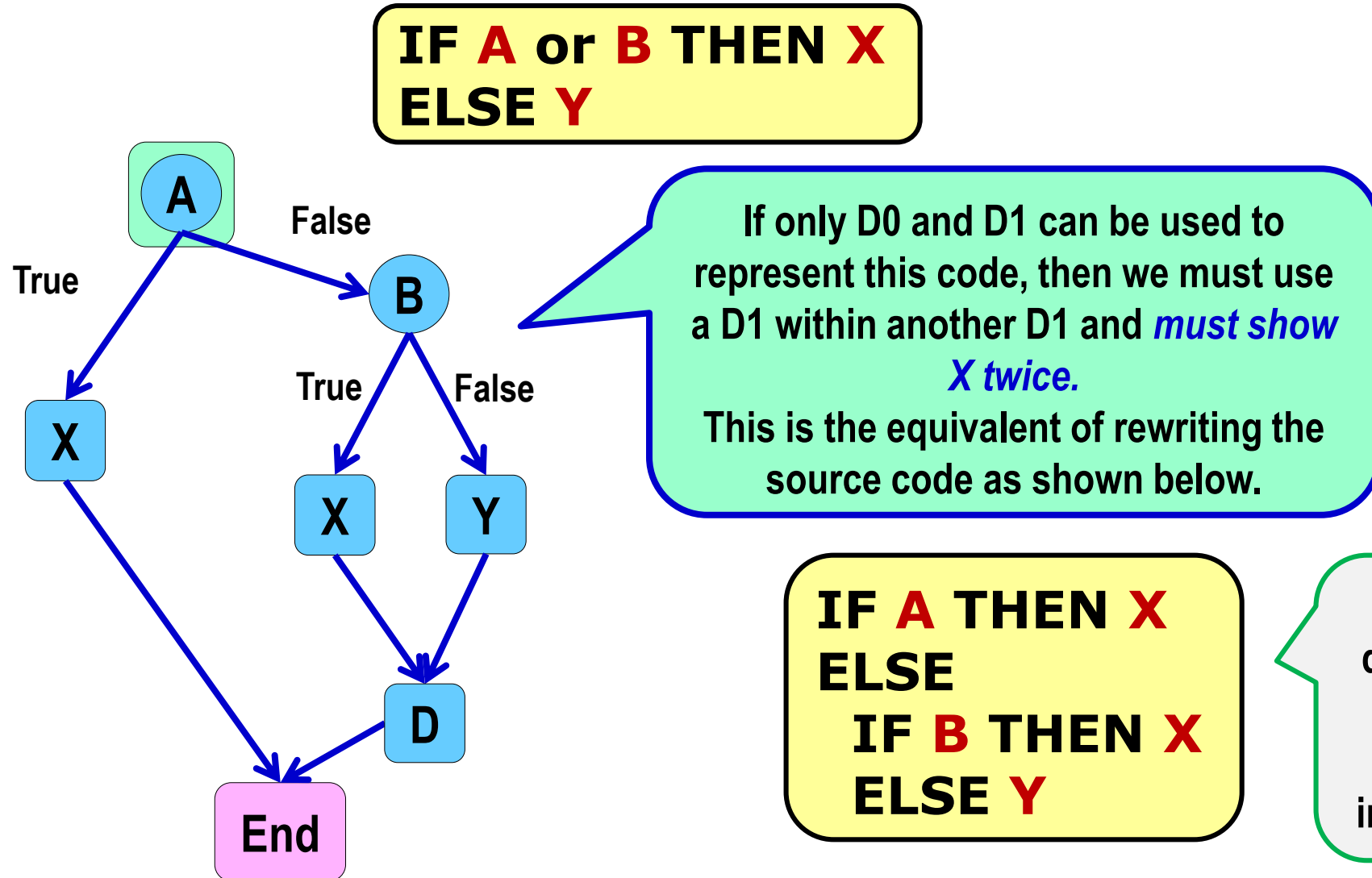
In other words, every organization defines *structured* in its own way.

Structured Program Flowgraphs: What Is Common and What Is Not

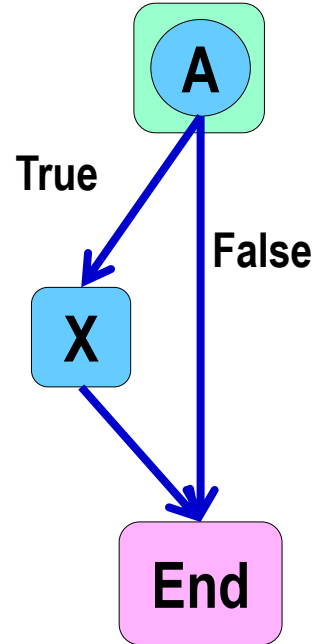
- **What all structured programs have in common**
 - Definitions of edges, nodes, etc.
 - Built out of the *three fundamental constructs*: sequence, selection, and iteration
 - It must be *possible to reduce* a program to a combination of a selected set, **S**, of prime flowgraphs
- **What is Different**
 - Which prime flowgraphs are included in the set **S**.

See Fenton, section 9.2 for a discussion of flowgraphs and structure and, in particular, section 9.2.1.2 for a *generalized notion of structuredness*.

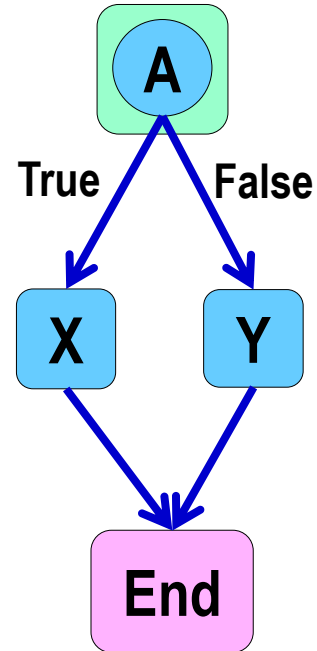
An Example of Why Additional Prime Flowgraphs are Useful



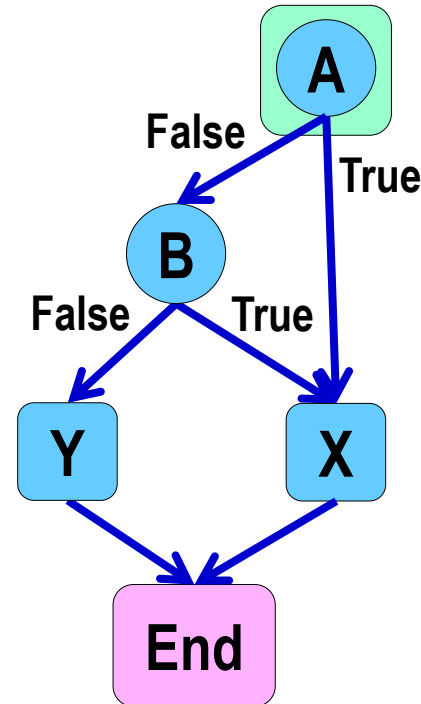
D₅ Was Introduced To Allow Common Boolean Selection Decisions



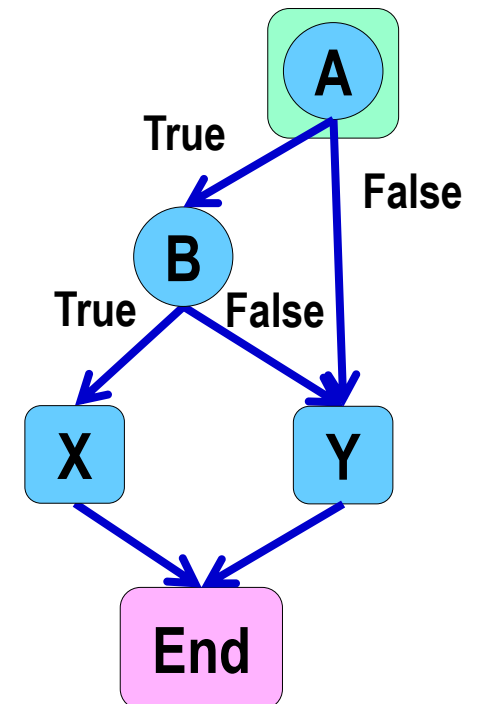
If A then B
(D₀)



If A then B else C
(D₁)

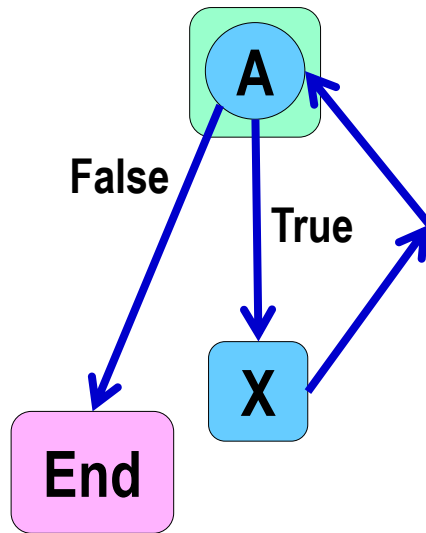


If A or B then X
else Y
(D₅)

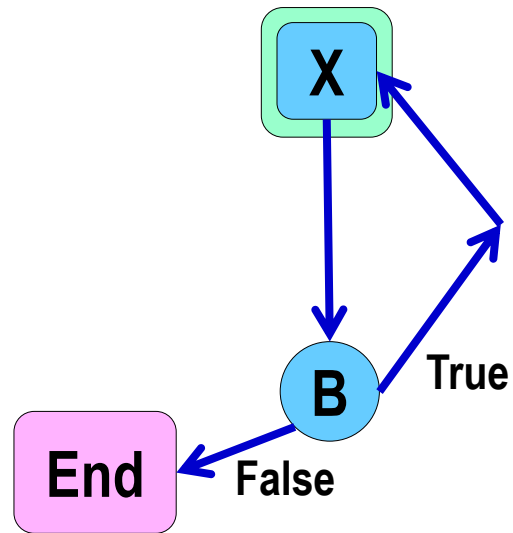


If A and B then X
else Y
(D₅)

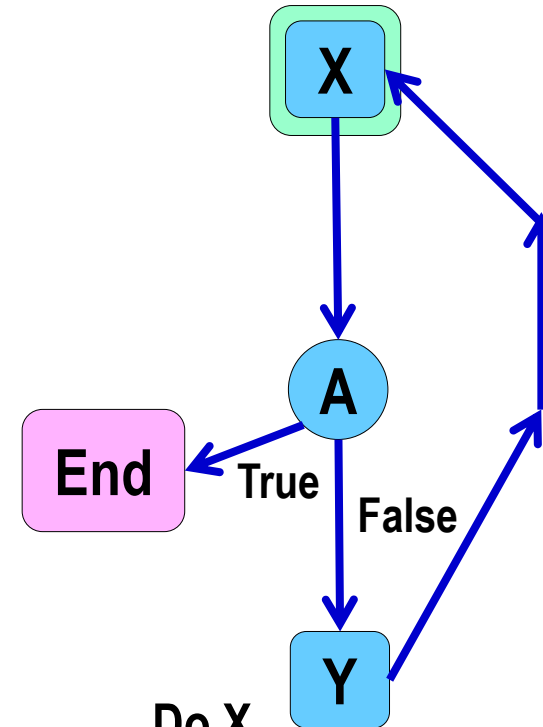
UTD D_4 Was Introduced to Allow Middle-Exit Loops



While A
Do X
(D₂)

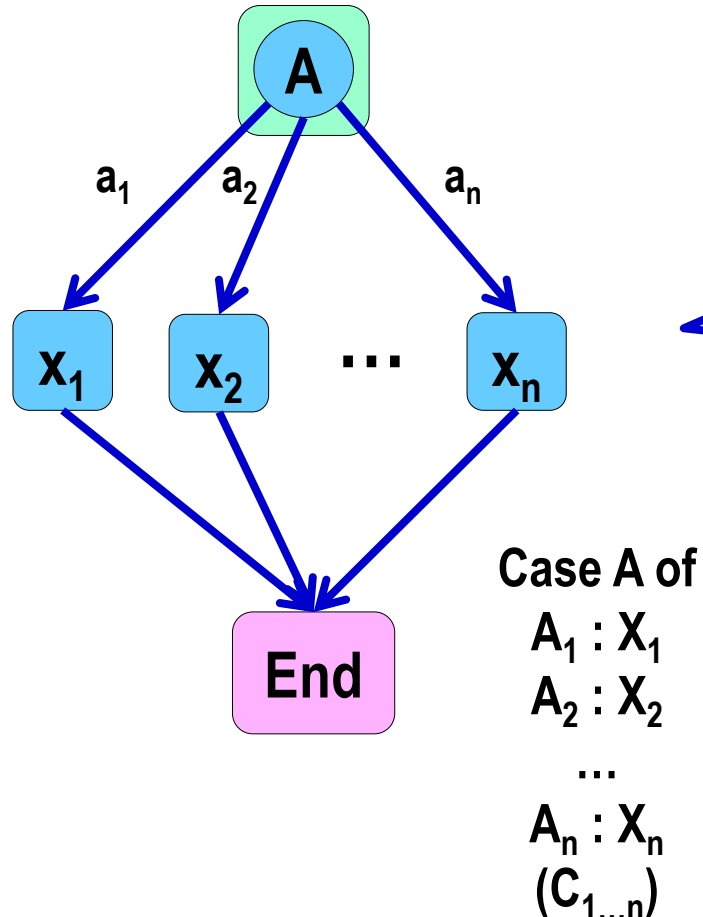


Repeat X
Until B
(D₃)



Do X
Exit when A
Do Y
Repeat
(D₄)

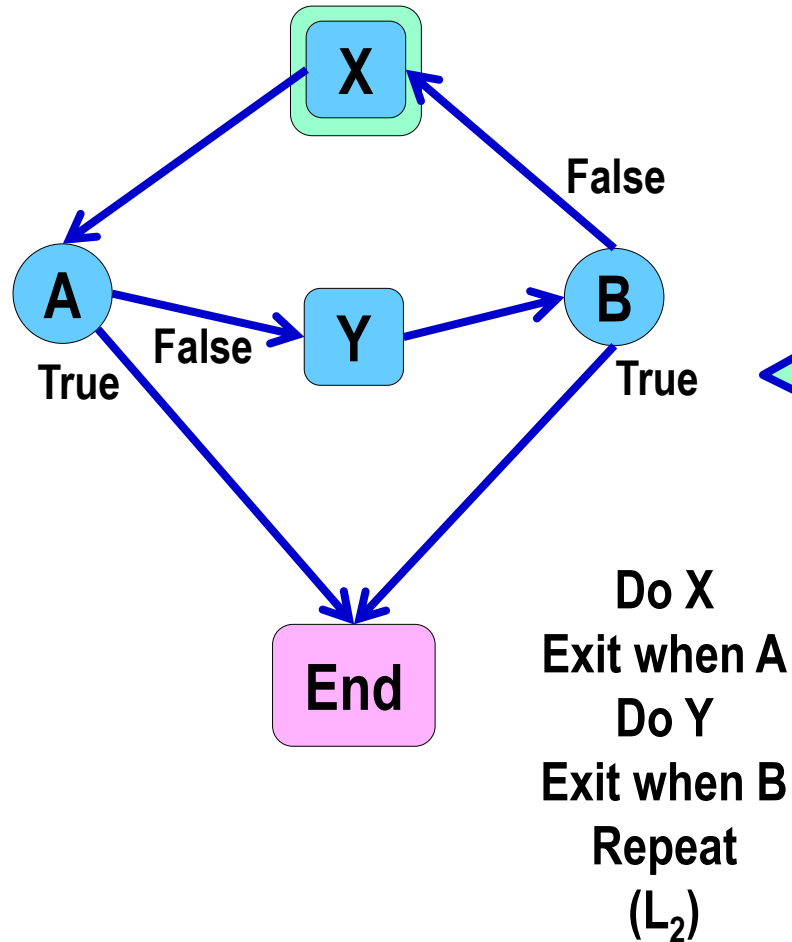
C Flowgraphs are Prime Flowgraphs for CASE Statements



Note that there are an arbitrary number of these, depending on n – the number of possible selections.

Note also that these are classified as “C” structured flowgraphs, not “D” structured flowgraphs, because, technically, the CASE statement is not one of the three fundamental control structures.

L Structured Flowgraphs Represent Multi-Exit Loops



A two-exit loop is shown (L₂). This is commonly used. However higher numbers of exits could be represented as well.

This also has its own classification (L) rather than being considered a D flowgraph because it is not one of the three fundamental control structures.

UTD Why Use Flowgraphs to Measure Complexity?

- **Directed Graphs clarify the flow of control between software elements**
- **Many measures of software complexity can be determined from directed graphs**
- **It is fairly easy to represent any program with a directed graph**
 - Note that there might be several ways to graph a program, but they should all have the same measure of complexity if they are done correctly

Combining Flowgraphs

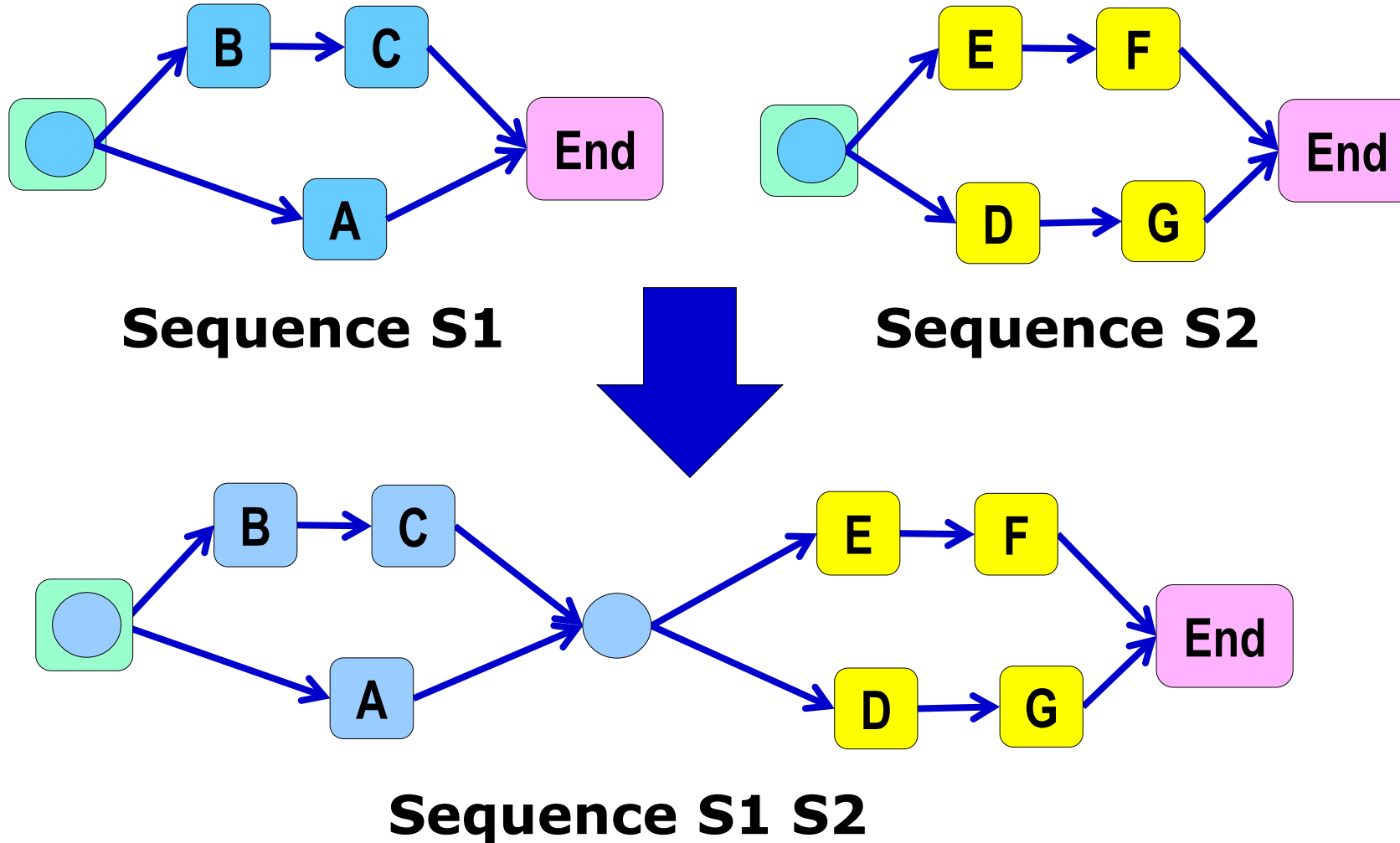
Flowgraphs with a single entry and single exit can be *combined* in the following ways:

- **Sequencing:** *Merging the end node* of one flowgraph *with the start node* of the other
- **Nesting:** *Replacing an arc* in one flowgraph *with the other flowgraph*

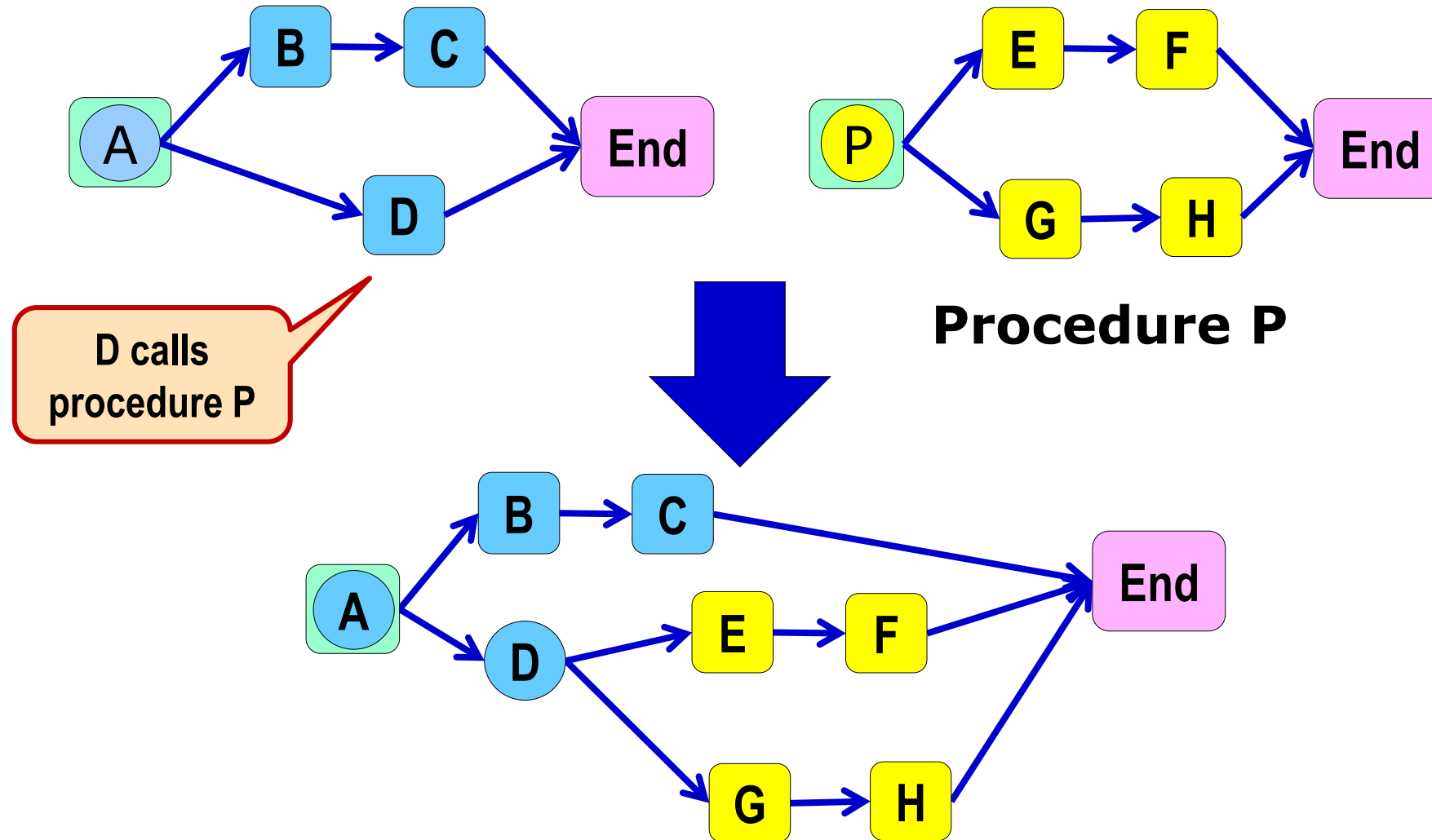
Flowgraphs can also be *reduced* or *condensed* or *decomposed* by reversing the above

- **For example, collapsing a nested flowgraph into a single node and arc**
 - This is, conceptually, the equivalent of replacing the nested flowgraph with a procedure call

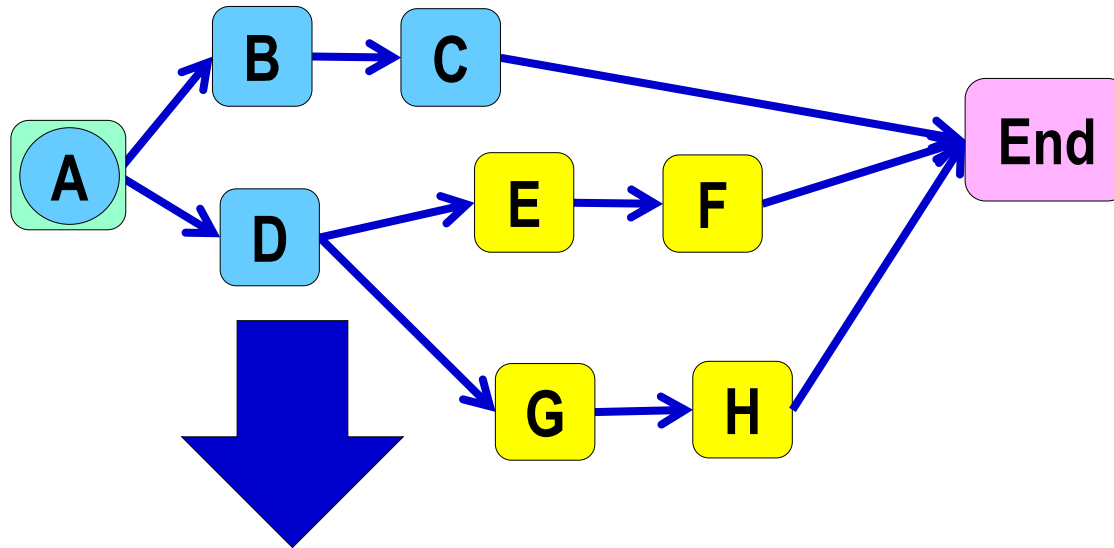
Sequencing Example



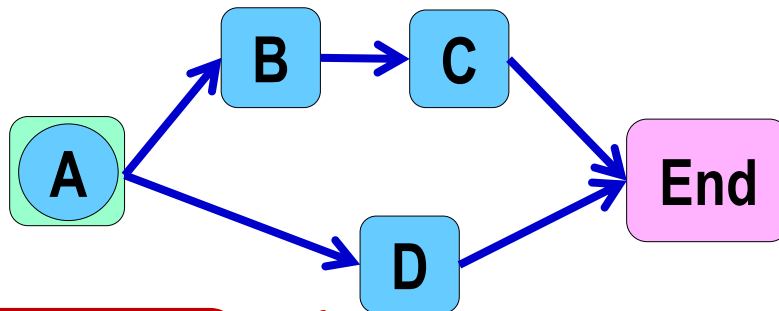
Nesting Example



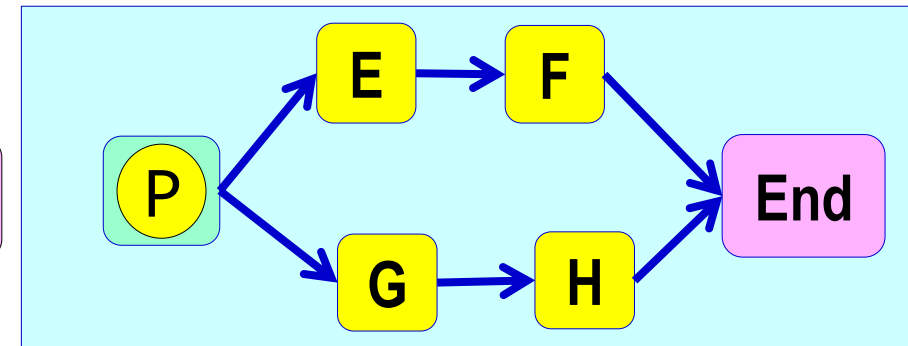
Reduction Example 1



Any single-entry, single-exit sub-graph can be replaced by a procedure call

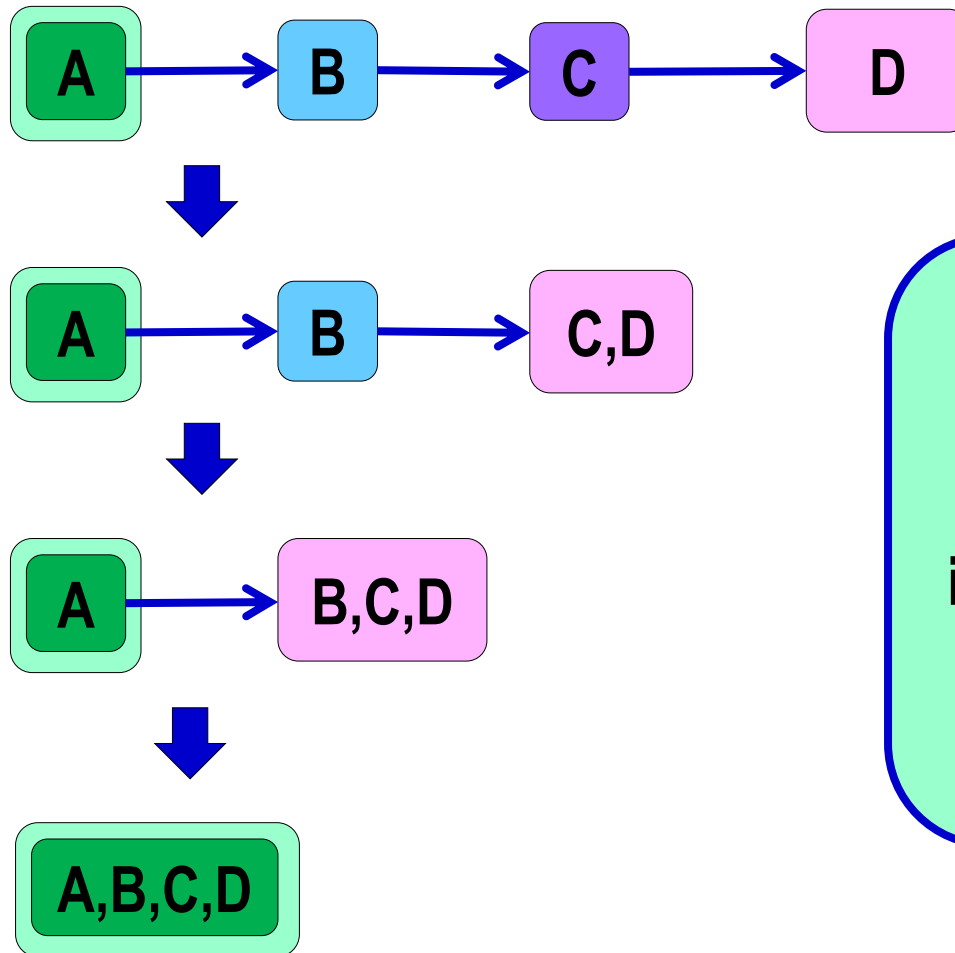


D calls procedure P



Procedure P

Reduction Example 2



Any sequence containing no decisions or iterations can be reduced to a single node

McCabe Cyclomatic Complexity

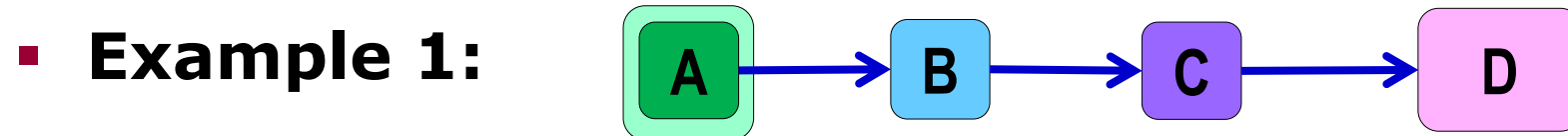
The ***Cyclomatic Complexity*** (**v**) of a Module or a System is:

- The number of **linearly independent¹ paths** (basis paths) through the module or system
- If F is a flowgraph², then **$v(F) = e - n + 2$**
 - Where **e** is the number of edges (arcs)
 - And **n** is the number of nodes
- If a system consists of multiple flowgraphs that are not connected together, the formula becomes:
 $v(F) = e - n + 2c$
 - Where **c** is the number of separate flowgraphs³

¹ To be discussed a little later ² With one entry and one exit

³ In graph theory these are called ***connected components***

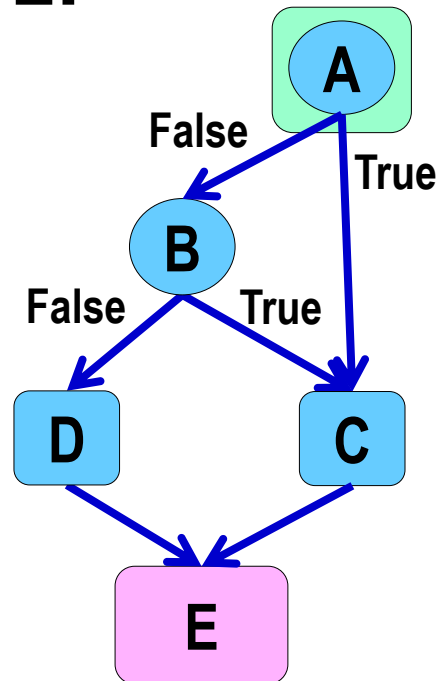
Examples of Cyclomatic Complexity



➤ $v(F) = e - n + 2 = 3 - 4 + 2 = 1$

➤ There is only 1 path through the code

■ **Example 2:**



➤ $v(F) = e - n + 2 = 6 - 5 + 2 = 3$

➤ There are 3 paths through the code:

- A B D E
- A B C E
- A C E

Why Is Cyclomatic Complexity Useful?

- Number of paths indicates *maximum number of separate tests* needed to test all paths
 - This should relate to the *difficulty of testing* the program
- It also indicates the *number of decision points in the program (plus 1)*
 - This should relate to the *difficulty of understanding and testing* the program

Cyclomatic complexity is not a perfect measure of these things (see Fenton, chapter 9) but it is a fairly reliable guide.

The Higher the Cyclomatic Complexity, the Harder the Code Is to Maintain

Cyclomatic Complexity

CC Value	Interpretation	Bad Fix Probability*	Maintenance Risk
1-10	Simple procedure	5%	Minimal
11-20	More complex	10%	Moderate
21-50	Complex	20% - 40%	High
50-100	“Untestable”	40%	Very High
>100	Holy Crap!	60%	Extremely High

*Bad Fix Probability represents the odds of introducing an error while maintaining code.

What Do We Mean by Linearly Independent Paths?

The ***number of linearly independent paths*** is the minimum number of end-to-end paths required to ***touch every path segment*** at least once.

- Sometimes the actual number of paths needed to cover the system is less than this because it may be possible to combine several path segments in one traversal.

There may be more than one set of linearly independent paths for a given flowgraph

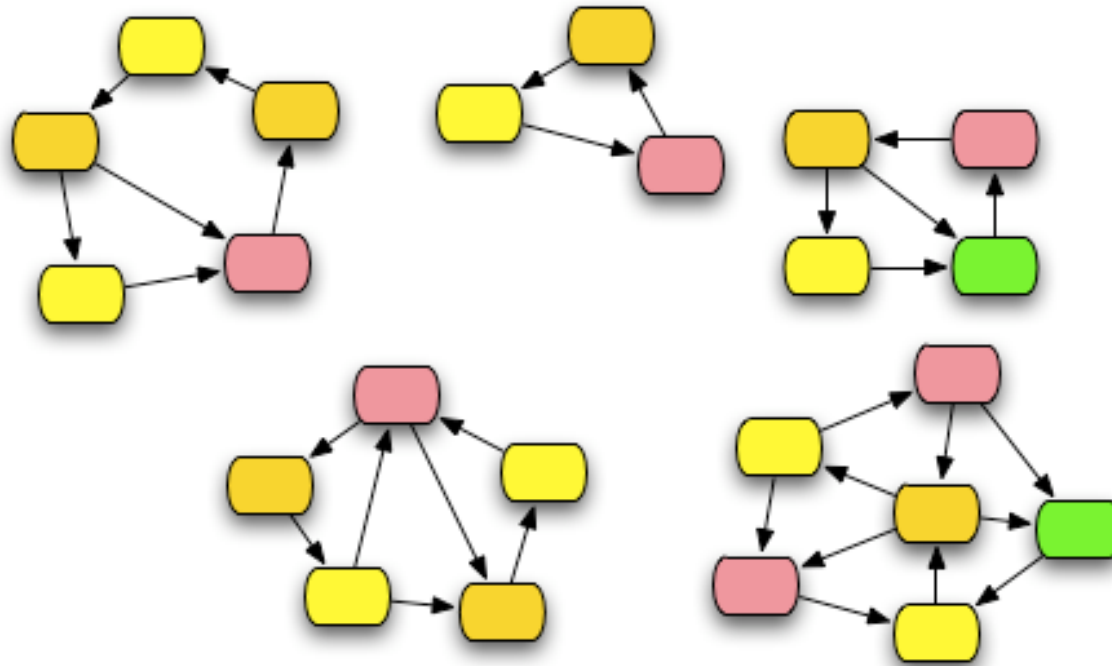
- More likely as you get more complex flowgraphs

Determining a set of linearly independent paths is something you might study in a course on testing or in a course on graph theory

- It gets harder as the cyclomatic complexity goes up

A Graph with Five Connected Components

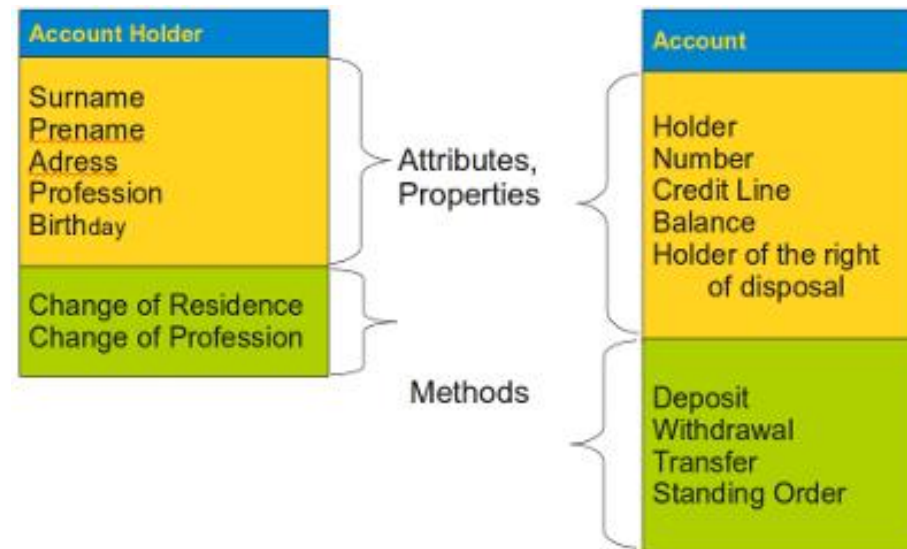
This graph has *five separate regions*, which are connected within themselves, but not to each other. Each region is called a *connected component*.



The graph above is not a flowgraph by our strict definition, because it has more than one start and stop node and not all nodes are connected to any given start or stop node. But it illustrates the concept of *connected components*.

Why Would We Care About Graphs with Many Connected Components?

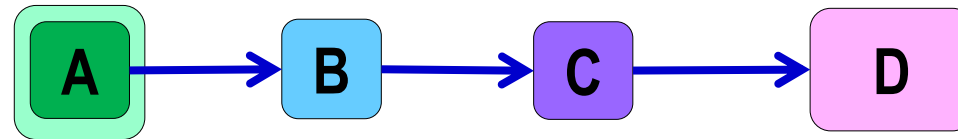
- We could measure the cyclomatic complexity of a *system consisting of several separate modules*
- In object oriented systems we could measure the cyclomatic complexity of a *class containing multiple methods*



McCabe Essential Complexity

The **Essential Complexity** (ev) of a Module or a System is:

- The cyclomatic complexity of the **fully reduced flowgraph**
- **Example:**



- **ev(F) = 1** because this can be reduced to one node

➤ **If the flowgraph is constructed completely of prime flowgraphs (i.e., it is structured) then the essential complexity will be 1.**

Essential complexity is intended to tell us *how well structured a program is*.

However

- **As originally defined, the only valid primes were the four D structured primes: D_0 , D_1 , D_2 , D_3**
 - So if you allow additional primes, do you revise the definition of essential complexity to include the new primes?
 - Do you allow D_4 and D_5 but nothing else?
 - What about the C structured primes and the L structured primes?

If your program is not “structured” it isn’t clear whether the essential complexity tells us much beyond that

- Does a larger essential complexity actually mean anything?
- If two programs have the same essential complexity, are they equally complex?
 - See fig. 9.13 in Fenton for an example
 - He shows two flowgraphs that have the same essential complexity, but intuitively one of them is a lot more complex and harder to understand than the other.

- Complexity: what and how to measure
- Structured Programs and Flowgraph Analysis
- Measures of Complexity
- ***Closing Remarks***

- **As we have seen, there are different ways to measure complexity**
- **Research shows that sometimes the attributes of complexity may conflict**
 - For example
 - low coupling doesn't always mean high cohesion
 - low cyclomatic complexity doesn't always mean easy to understand
 - structured software may be awkward to produce in languages without certain constructs

Use complexity measures as guidelines, not as “magic numbers” that result in rigid requirements for code.

END OF Part 5

Any Questions?



End of Lecture

References

Part 3 (1 of 2)

Chatfield, C., *Statistics for Technology, A Course in Applied Statistics, Third Edition*, Chapman and Hall, London (1983), ISBN 978-0412253409.

Fenton, Norman and James Bieman, *Software Metrics: A Rigorous and Practical Approach, Third Edition*, Chapman and Hall, 2014. ISBN 978-1439838228, Chapter 6.

Hedstrom, John and Dan Watson, “Developing Software Defect Prediction,” *Proceedings, Sixth International Conference on Applications of Software Measurement*, 1995.

Jones, Capers, *Applied Software Measurement*, McGraw Hill, 1991. ISBN: 0-07-032813-7.

Knuth, Donald, *Seminumerical Algorithms: The Art of Computer Programming, Vol II*, Addison-Wesley, 1969. ASIN: B00157WFAU

References

Part 3 (2 of 2)

Ott, R.L. and M. T. Longnecker, *An Introduction to Statistical Methods and Data Analysis, 6th Edition*, Duxbury Press (2008), ISBN 978-0495017585.

Snyder, Terry and Ken Shumate, *Defect Prevention in Practice* (Draft white paper), Hughes Aircraft Company, October 22, 1993.

Ross, Sheldon M.. *Introduction to Probability Models*, Academic Press, 1993. Musa, John, *Software Reliability Engineering: More Reliable Software, Faster Development and Testing*, McGraw Hill. ISBN: 0-07-913271-5.

References

Part 4 (1 of 2)

Abran, A., et. al., "Functional Complexity Measurement", *Proceedings, IWSM 2001 - International Workshop on Software Measurement*.

Chidamber, S. and Chris Kemerer, *A Metrics suite for Object Oriented Design*, MIT Sloan School of Management E53-315 (1993).

Fenton, Norman and James Bieman, *Software Metrics: A Rigorous and Practical Approach, Third Edition*, Chapman and Hall, 2014. ISBN 978-1439838228, **Chapter 9**

Fenton, N. and A. Melton, "Deriving Structurally Based Software Measures," *Journal of System Software*, vol 12 (1990), pp 177-187.

Henry, S. and D. Kafura, "Software Structure Metrics Based on Information Flow", *IEEE Transactions on Software Engineering*, Volume SE-7, No. 5 (Sept, 1981), pp 510-518.

References

Part 4 (2 of 2)

IEEE 9982.2 (1988). *IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software, A25. Data of Information Flow Complexity.* P112.

Stevens, W., G. Myers and L. Constantine, "Structured Design", *IBM Systems Journal*, vol 13, no 2 (1974), pp 115-139.

Kitchenham, B. A., "Measuring to Manage", in Mitchell, Richard J. (editor), *Managing Complexity in Software Engineering*, London, Peter Peregrinus, Ltd. (1990). ISBN 0 86341 171 1

Lavazza, L. and G. Robiolo, "Functional Complexity Measurement: Proposals and Evaluations", *Proceedings of ICSEA 2011: the Sixth International Conference on Software Engineering Advances.*

References

Part 5

Dijkstra, Edsger, "GO TO Considered Harmful", letter to the editor of *Communications of the ACM*, March, 1968.

Fenton, Norman and James Bieman, *Software Metrics: A Rigorous and Practical Approach, Third Edition*, Chapman and Hall, 2014. ISBN 978-1439838228, Chapter 9

Fenton, N. and A. Melton, "Deriving Structurally Based Software Measures," *Journal of System Software*, vol 12 (1990), pp 177-187.

McCabe, Thomas, "A complexity measure," *IEEE Transactions on Software Engineering*, vol SE-2, issue 4 (December, 1976), pp 308-320.

Stevens, W., G. Myers and L. Constantine, "Structured Design", *IBM Systems Journal*, vol 13, no 2 (1974), pp 115-139.

Exercise

Given a Program, Determine its Flowgraph and its Cyclomatic Complexity

- **Review the program (handout) and draw its flowgraph**
- **Compute the Cyclomatic Complexity**
- **Report to the Class**

- **Option:**
 - One or more students may be asked to explain their flowgraph to the class