Software Engineering

Dokuz Eylül University

14 March – 26 May, 2000

Chapter 1

Introduction

□ This lecture will be based on the book:

- S. L. Pfleeger:

  Software Engineering: Theory and practice.

  Prentice Hall, 1999.

□ Contents:

1. What is Software Engineering?
2. Why Software Engineering?

3. Planning and Managing the Project.
4. Capturing the Requirement.
5. Designing the System.
6. Writing the Programs.
7. Testing the Programs.
8. Testing the System.
9. Delivering the System.

10. Evaluating Products.
11. Improving Processes.

□ What is Software Engineering?

- An engineering that helps us

  to design & develop "*high quality software.*"

- What then is a "*high quality software*?"
  - more efficient;
  - more precise;
  - easier to modify;
  - easier to use;
  - easier to understand.

- The term "*Software Engineering*" was first used in 1968 at a NATO conference.

□ Why Software Engineering?

- Badly developed system
  would cost more than properly developed.

- Improperly designed or programmed software
  would threaten our lives.

- Sometimes it is
  impossible to test.

□ Example 1-1:

In 1985, Internal Revenue Service in US
hired Sperry Corporation to build
an automated federal income tax form processing system
$103,000,000

$\Downarrow$

proved inadequate
(**running cost was nearly twice** what was expected)

$\Downarrow$

had to be replaced soon
extra **$90,000,000** was needed!

$\Downarrow$

a delay to return refunds to the taxpayers caused
$40,200,000 (interest) & **$22,300,000** (overtime wages)

K. Sawyer (1985) “The mess at the IRS.”
Washington Post National Weekly Edition,
November 11, pp.6–7.

$\Downarrow$

In 1996, it had not improved yet

Congressman J Lightfoot’s words:
“**A $4-billion fiasco**
that is floundering because of inadequate planning.”

Los Angels Times, March 29 (1996)

□ Example 1-2:

On June 4 1996,

the Ariane-5
(a space rocket belonging to the European Space Agency)

veered off course 40 seconds after being launched and performed perfectly,

due to an *error* of the embedded software.

$\Downarrow$

It was destroyed by remote control.

$\Downarrow$

The total cost of the disaster was **$500,000,000**.

J. L. Lions et al. (1996)
"Ariane 5 Flight Failure."
Report by the inquiry board,
European Space Agency.

□ Example 1-3:

- Y2K

□ Example 2-1:

THERAC-25
(a radiation therapy machine)

$\Downarrow$

the software designers had not anticipated
the use of "arrow keys" in non standard way

$\Downarrow$

issued a highly concentrated dose of radiation
when low levels were intended.

$\Downarrow$

killed several patients.

□ Example 2-2:

In December 1995,

An American Airline's jet

crashed in Colombia;

killed all but 4 of the 163 people aboard.

⇑

One-letter code for "Cali" was the same as the one for "Bogota"
132 miles in the opposite direction.

Washington Post,
August 24, 1996

(Associated Press)

□ Example 2-3:

In 1996,

Two police officers
in the Lothian and Borders region of Scotland
were using a radar gun
to identify speeding motorists on the Berwickshire moors.

Suddenly, their radar gun locked up,
with a speed indication of over 300 miles per hour.

Seconds later, a low-flying Harrier jet flew by.

The Harrier's target seeker had recognized the radar
and thought it to belong to an enemy.

(Fortunately, the Harrier was operating unarmed,
since normal behavior would have triggered
only an automated retaliatory missile!)

Pilot magazine, 1996

□ Example 3:

- Strategic Defense Initiative (US)
  - proposed by President Regan (1985).
  - at least 10,000,000 lines of code.

- Space Shuttle (US)
  - computers on the ground ⇒ 3,000,000,000 lines of code.
  - shuttle ⇒ 100,000 lines of code. [1]

$$\Downarrow$$

no way of testing the code, line by line.

- Atomic Power Plant

$$\Downarrow$$

10^9 safety critical

[1] B. Rensburger (1985) "The Software is too Hard. Washington Post National Weekly Edition, November 11, pp.10-11."

□ Safety Critical:

System in which *failure* poses a threat to life or health should have a reliability of at least 10^9

$$\Downarrow$$

System can fail no more often than once in 10^9 hours of operation.

□ Fault, Error, and Failure

- A **fault** occurs when a human makes a mistake, called **error**, in performing some software activity.

- A **failure** is a departure from the system's required behavior.

□ Wasserman's discipline of Software Engineering:

- A. I. Wasserman:

  "Toward a Discipline of Software Engineering."
  Software, Vol.10, No.6, pp23–31, IEEE, 1996.

$\Downarrow$

Why software development today is different
from 10 or 20 years ago.

□ **7 factors that have changed software engineering:**
Wasserman (1996)

- criticality of *time*
- lower hardware costs and
  greater development and maintenance cost
- powerful PC
- networking
- object-oriented technology
- GUI
- unpredictability of waterfall model

$\Downarrow$

Shift from the traditional "Waterfall Model"
to the modern ones.

□ Wasserman's discipline of Software Engineering:

- try *abstraction* of the problem
- build *models*
- produce *prototyping*
- make *architecture* of a system clear
- evaluate *developing process*
- make it possible to *reuse* items from previous development
- *quantify* improvement
- establish tools

Chapter 2

Planning and Managing the Project

□ Planning the Project:

- How much the project will cost

and

- how long the project will take?

□ Divide and Conquer Strategy:

(Analysis and Synthesis)

(1) Break the problem into its component parts.

(2) Devise a solution for each part.

(3) Put the pieces together to form a coherent whole.

□ **To apply the strategy to planing the project:**

- By examining the project carefully

$\Downarrow$

- we can separate development into a succession of **phases.**

$\Downarrow$

- Each phase is composed of **steps.**

$\Downarrow$

- Each step can be subdivided further into **activities.**

□ Process Model:

A **process model** also provides

a view of the **phases** and **steps.**

$\Downarrow$

We will be back this issue later in detail.

□ Activity and Milestone

- Developers can point to
  - **activities**
    - ∗ to show what work is under way.
  - **milestones**
    - ∗ to show what works have been completed.

□ Activity and Milestone?

- A **activity:**
  - a part of the project that takes place *over a period of time.*

- **milestone:**
  - the completion of an activity: *a particular point in time.*
  - indicates (to developers and customers) that a measurable level of progress has been made, e.g.,
    - completion of the user's manual;
    - performance of a given set of calculations;
    - demonstration of the system's ability.

Note:
Each **activity** is characterized by a milestone.

□ 4 parameters to describe an activity

- **precursor:**
  - an event (or set of events) that must occur before the activity starts, or
  - a conditions that allows the activity to start
- **duration:**
  - time needed to complete the activity
- **due date:**
  - date by which the activity must be completed
- **end point:**
  - a milestone or *deliverable*

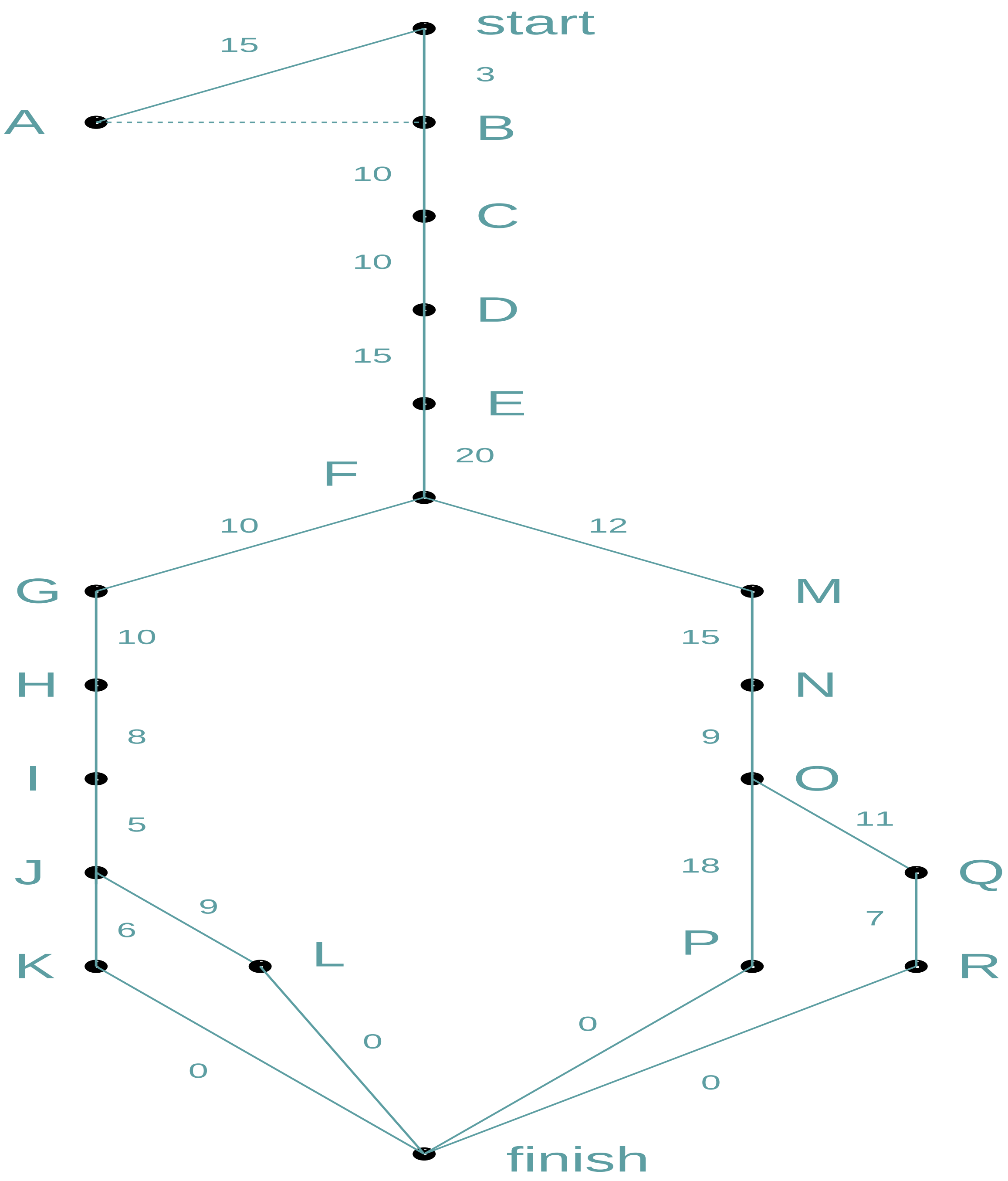
□ **deliverable:**

- **documents**
- **demonstrations of:**
  - function
  - subsystems
  - accuracy
  - reliability
  - security
  - performance

□ Activity Graph:

- relationships among activities expressed by the 4 parameters.
  - node ⇒ the project milestones
  - arch ⇒ activities

An Example of Activity Graph



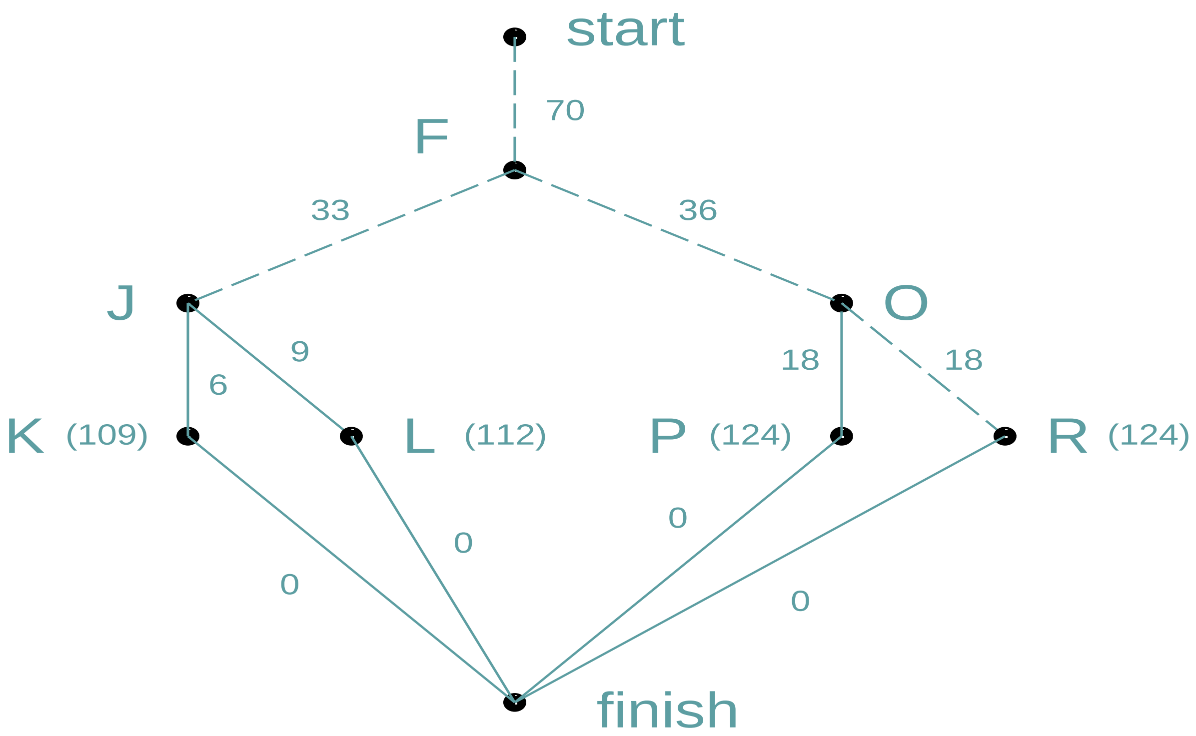
□ Slack Time:

{Slack Time}
= {Available Time} – {Actual Time}

or

= {The latest time an activity may begin without delaying the project} – {The earliest time the activity may begin}

NOTE:
the longest path $\Leftarrow$ {Slack time} $= 0$


start
70
F
33
36
J
O
9
6
18
18
K (109)
L (112)
P (124)
R (124)
0
0
0
0
finish

Since **critical path** has no slack,

$\Downarrow$

there is *no margin for error.*

$\Downarrow$

Thus the **activity graph**
helps us to understand
the impact of a schedule slippage.

Critical Path Method also

shows us

- the minimum amount of time it will take to complete the project;

reveals

- activities that are most critical to completing the project on time.

NOTE:
There may be more than one critical path.

The Longest Path

- has a slack of zero for each of its nodes (activities);

- determines whether or not the project is on schedule.

$$\Downarrow$$

- That's why this is called the *critical path*.

□ The Shortest Path Algorithm.

Notation:
S: the node to start with, T: The goal;
$E(U)$: energy of node U (the shortest distance from S)
$e(U)$: energy of node U

Step 1

- Assign energy zero to S: $E(S) = 0$.
- Assign label ∞ to all nodes other than S: $e(S) = \infty$.

Step 2

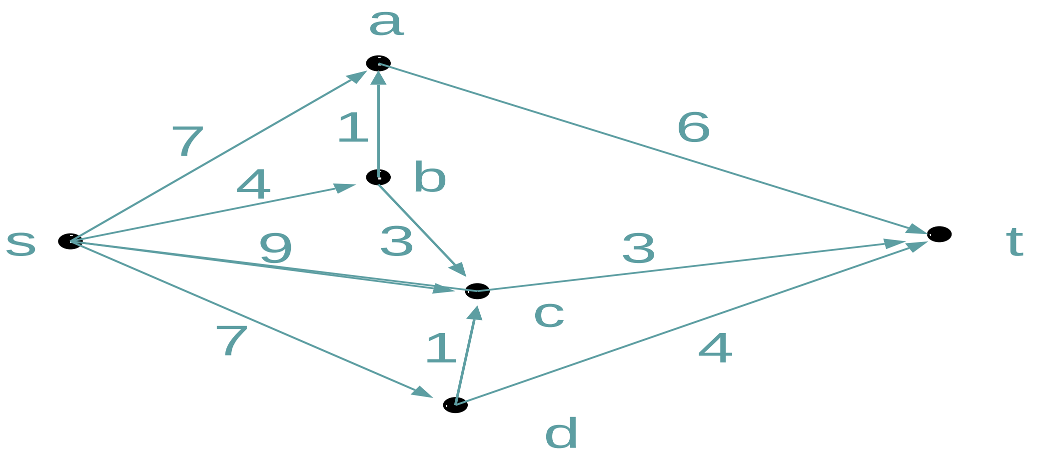
- FOR each of the nodes U_i that was already assigned an energy
  - FOR each V_j that are directly reachable from U_i
    - IF $E(U_i) + r(U_i, V_j) < e(W_j)$
      THEN $e(V_j) = E(U_i) + r(U_i, V_j)$
  - Find a minimum value of labels among nodes that were not assigned an energy yet, and assign energy of that value to the point(s).

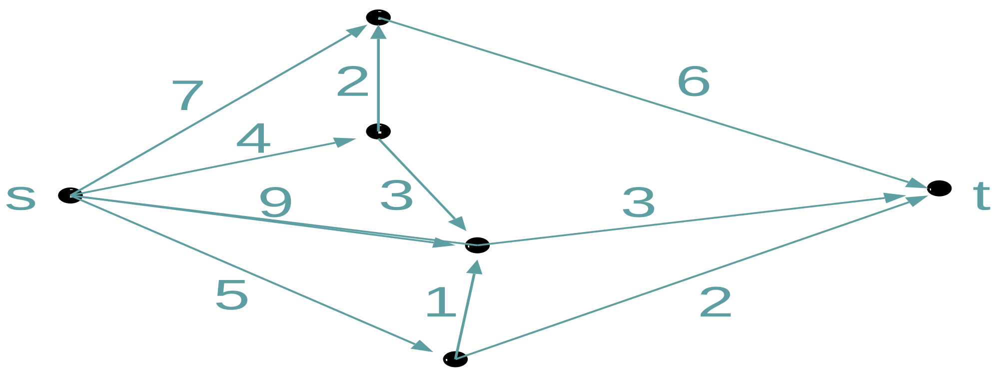
Step 3

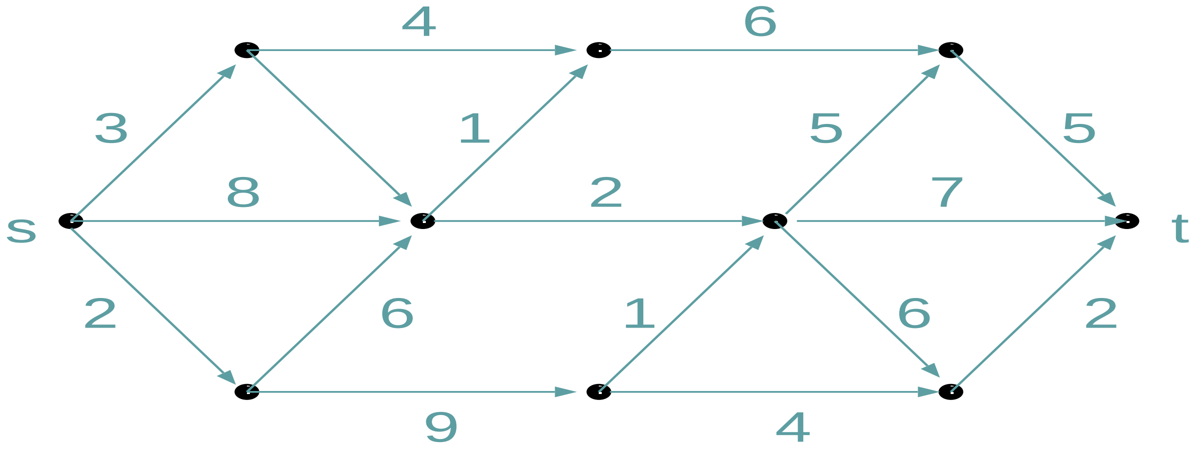
- Repeat step 2 until all nodes are assigned an energy.

Step 4

- Pick up arcs when $E(A) - E(B) = r(A, B)$


a
b
s
c
t
d
7
4
9
7
1
3
1
6
3
4


s
t
7
4
9
5
2
3
1
6
3
2


s
t
3
8
2
4
1
6
9
2
6
5
1
7
6
4
5
2

The above example is too simplified, however.

- What happens if the activity graph has loops?

  e.g. requirements to be respecified

- Durations are always clear to be exact?

  → To each activity we can assign a probable duration according to some probability distribution, so that each activity has an expected value and a variance.

  → *Gaussian distribution* instead of exact duration

  → Program Evaluation and Review Technique (PERT).

  e.g. Those activities most likely to be bottlenecks can be identified.

  * F. S. Hiller and G.J. Liberman (1967)
  Introduction to Operations Research.
  Holden-Day, San Francisco, US.

Exercise

1 List its precursors and compute the activity's earliest start time, the latest one, and the slack time.

2 Find the critical Path.

□ Critical Path Method (CPM)

- Who must wait for what
  as the project is being developed?
- Which activities must be completed on schedule
  to avoid delay?

□ Estimation:

- Time (Schedule) Estimation
- Cost Estimation
- Effort Estimation

$\Downarrow$

affect *resource allocation* and *project feasibility*;

$\Downarrow$

must be done *as early as possible*;

$\Downarrow$

must be repeated throughout the project.

□ Cost Estimation:

- How much the project is likely to cost?
  - Cost overruns can cause customers to cancel projects.
  - Cost underestimate can force a project team to invest much of their time without financial compensation.
  - A good cost estimate helps the project manager to know how many developers will be required and to arrange for the appropriate staff to be available when they are needed.

□ Cost for what?

(1) Facilities cost for:

- items that provide the physical environment in which the developers will work ⇒ well-understood.
  - hardware, space, furniture, telephones, modems, heating and air-conditioning, cables, disks, paper, pens, photocopiers, etc.

- new environment to be created ⇒ difficult to estimate.
  - a security vault, a raised floor, temperature or humidity controls, or special furniture.

- other hidden cost
  - e.g., a programmer needs a space
    - a minimum of 100 square feet of floor
      with 30 square feet of work space;
    - quiet, free from telephone call, uninvited visitors,
    - in order to be effective.

(2) Software and tools to support development efforts.

– to design and code the system;

– to capture requirements;

– to organize documentation;

– to test the code;

– to keep track of changes;

– to generate test data;

– to support group meetings;

– and more.

$\Downarrow$ called

Computer-aided Software Engineering (CASE)

□ Effort Estimation:

- How many *staff-days* of effort will be required to complete the project?

$\Downarrow$

Cost component with the greatest degree of uncertainty.

In a project,

- work style
- how team members are organized
- ability
- interest
- experience
- training
- other employee characteristics

$\Rightarrow$ affect the time it takes to complete a task.

More over, when a group of workers must communicate and consult with one another,

- **effort**

$\Rightarrow$ is required to shorten the time for meeting, documentation, training and so on.

□ How can we estimate it?

- Judgment by Experts
- Algorithmic Way

□ Expert Judgment:

- informal technique that depends on
  - competence
  - experience
  - objectivity
  - perception of estimator

$\Downarrow$ Efforts to be less informal.

(1) Using analogy of the past

(2) Delphi Technique

(3) Software Cost(Effort) Matrix

□ (1) Analogy from previous experiences:

- ask several experts to estimate

$\Downarrow$

"This team is similar to that team, so the point is $\cdots$"

$\Downarrow$

· a pessimistic prediction $\Rightarrow (x)$

· an optimistic one $\Rightarrow (y)$

· a most likely guess $\Rightarrow (z)$

$\Downarrow$

$(x + 4y + z)/6$

(the means of the beta probability distribution)

This relies too much on the experts' ability to determine which projects are similar and in what ways.

□ (2) Delphi Technique:

- Experts are asked to make predictions *secretly.*

$\Downarrow$

Average estimate is calculated and presented to them.

$\Downarrow$

Again, they are asked to revise his/her estimate.

$\Downarrow$

The average when no one revised any more is adopted.

□ (2+1) Variations of Delphi:

- They discuss when their estimates are feed-backed.
- Points of each expert are circulated anonymously among them.

□ (3) Software Cost Matrix: [2]

- row ⇒ The type of software.
  - Control
  - Input/Output
  - Pre/Post processor
  - Algorithm
  - Data management
  - etc.

- column ⇒ difficulty
  - Old (O) or New (N) ?
  - Easy (E), Modest (M), or Hard (H)?

- entry ⇒ cost per line of code

[2] R.W.Wolverton (1974) "The Cost of Developing Large-scale Software." IEEE Computer, 23(9) pp.8–24.

type of software	Difficulty					
	OE	OM	OH	NE	NM	NH
Control	21	27	30	33	40	49
Input/output	17	24	27	28	35	43
Pre/post processor	16	23	26	28	34	42
Algorithm	15	20	22	25	30	35
Data management	24	31	35	37	46	57

$\Downarrow$

Though this example is based on specific company (TRW: Software developing company in US),

$\Downarrow$

applicable to your environment.

□ Example of using software cost matrix:

- A system with three modules.
  - Input/Output module that is old and easy (100 lines).
  - Algorithm module that is new and hard (200 lines).
  - Data management module that is old and moderate (100 lines).

$$\Downarrow$$

$$100 \times 17 + 200 \times 35 + 100 \times 31 = 11.800$$

☐ Pros and cons:

- Subjective;
- Non-linearity
  - Two people cannot produce code twice as fast as one, but extra time may be needed.
    * communication
    * coordination
    * accommodation of differences in interest, ability, experience, etc.
- Too dependent on data;
- Too simple.

$\Downarrow$

Algorithmic Method

□ Algorithmic Method:

Relationship between effort and the factors that influence it.

$\Downarrow$

Effort E is represented as a function of these factors:

$$E = f(x_1, x_2, \cdots, x_n).$$

where, e.g.,

x_1 = project size
x_2 = experience
x_3 = application type
$\cdots\cdots$

□ Most of the models acknowledge that

project size (S) is most influential.

$$\Downarrow$$

$$E = (a + bS^c) \cdot m(\mathbf{X}).$$

where

$$\mathbf{X} = (x_1, x_2, \cdots, x_n).$$

(1) **Walston et al. (1977)** [3]

...One of the first of such models.

$$E = 5.25S^{0.91} \cdot m(\mathbf{X}).$$

Size was measured as lines of code including comments (if they did not exceed 50% of the total lines.)

$\Uparrow$ $m(\mathbf{X})$ is modified by 29 factors

1. Number of pages of documentation per 1000 lines of codes.
2. Ratio of average staff size to project duration (people per month)
3. Previous experience with applications of similar size and complexity
4. Previous experience with the programming language

   ...

29. Overall complexity of code

[3] Walston et al. (1977) "A Method of Programming Measurement and Estimation." IBM Systems Journal, 16(1) pp54–73.

Walston et al.'s methods was applied to IBM's

- the data built systems
- with 4000 – 467,000 lines
- written in 28 different languages
- on 66 computers.

$\Downarrow$

12 – 11,758 person-month effort

In conclusion

$$E = 5.25 S^{0.91} \cdot m(\mathbf{X}).$$

where

$$m(\mathbf{X}) = \sum_{i=1}^{29} c_i \cdot w_i,$$

where

$$c_i = \begin{cases} 1 & \text{if the factor increases productivity,} \\ 0 & \text{if no effect,} \\ -1 & \text{if it decreases productivity,} \end{cases}$$

and w_i represents importance of each factor, i.e.,

$$\sum_{i=1}^{29} w_i = 1$$

(2) **Bailey et al. (1981)** [4]

proposed a *meta-model* for building estimation equation that reflects your own company's characteristics.

$$E = 5.5 + 0.73 \cdot X^{0.91} \cdot m(\mathbf{X}),$$

where $m(\mathbf{X})$ is modified with

- methodology (9 items; 45 points)
- complexity (7 items; 35 points)
- experience (5 items; 25 points)

judged by the project manager.

They demonstrated the model using a data base of 18 projects written in Fortran at NASA.

$\Downarrow$ after minimization of the error

it was found that the result was very accurate.

[4] Bailey et al. (1977) "A meta-model for Software Development Resource Expenditures." Proceedings of the International Conference on Software Engineering. pp.107-116.

Adjustment is as follows:

$$m(\mathbf{X}) = \begin{cases} R & \text{if} R \geq 1, \\ 1/(2 - 1/R) & \text{if} R < 1, \end{cases}$$

where

$$R = \frac{\text{actual effort}}{\text{predicted effort}}.$$

(3) **Boehm (1981)** [5]

$$E = bS^c\dot{m}(\mathbf{X}).$$

Modification is somewhat more sophisticated, dividing development into 3 periods.

[5] B.W.Boehm (1981) Software Engineering Economics. Plentice Hall.

- Stage 1
  - the model acknowledges that lines of code are impossible to know;
  - instead, they use *number of server and client data tables*, and the percentage of *reusable parts from previous projects* (*object points)*.
  - c is set to be 1.
- Stage 2
  - still not enough information as to size;
  - size is estimated by degree of
    - reuse
    - breakage
    - maintenance

    (*function-point calculation*)
  - $c \in [1.01, 1.26]$ depending on
    - the degree of novelty of the system
    - early architecture and risk resolution
    - team cohesion
    - process maturity.
- Stage 3
  - sizing can be done in terms of *line of codes.*

□ Risk Management

- What is a risk?
  - An unwelcome events during development/maintenance.

- Risk Management.
  - Plans to avoid, or if inevitable, minimize them.

□ Risk Impact

... A loss associated with a risk.

- A loss of time, quality, money, control, understanding, etc.

e.g.,
A dramatical requirement change after the design is done.

$\Downarrow$ What happens?

- loss of control and understanding
  if the new requirements are unfamiliar.
- losses of time and money
  if the design is not flexible.

□ Risk Probability

... the likelihood that a risk will occur.

e.g.,
When a system that is being developed on one machine will be ported to another when the system is fully tested.

$\Downarrow$

The likelihood that the new machine will not delivered by the vendor

Note:
probability = 1
$\Downarrow$
the risk is called a **problem**

□ Risk Control

... What we can do to minimize or avoid the impact?

- A set of actions taken to reduce or eliminate a risk.

$\Downarrow$ e.g.,

- If the requirements may change after design,

$\Rightarrow$ we should create a flexible design.

- If the 2nd machine is not ready,

$\Rightarrow$ we may identify other models that have the same functionality and performance and run the software until the new model is delivered.

□ Quantification of the effect of a risk.

- {Risk Impact} × {Risk Probability} ⇒ {Risk Exposure}

$\Downarrow$ e.g.,

- The cost to redesign to new requirements = \$50,000;
- The likelihood of the requirements change after design = 0.3.

$\Downarrow$ then

Risk Exposure = \$ 15,000.

Note:
Risk Probability and risk impact can change over time.

$\Downarrow$

The project manager should track them over time.

□ Two major sources of risk.

- Generic risks
  - misunderstanding the requirements
  - losing key personnel
  - insufficient time for testing
  - etc.

- Project specific risks
  - a software needed for the development will not be ready on time.
  - etc.

□ Risk Management Activities.

- Risk Assessment
  - understanding what may occur during development and maintenance.

- Risk Control

  (later in detail)

□ Risk Assessment.

- Identify the risks
  - Check list
    - If the system you are building is similar to previous one, you have a checklist of problems that may occur;
    - If it's new, you may augment the check list with analysis.
  - Decomposition
    - By decompose the process into small pieces, you may anticipate problems that may arise.
  - Assumption and Decision analysis
    - Analyze your assumptions and decisions you made, as to:
      - how the project will be done;
      - who will do it;
      - with what resources.

        (each one is assessed to know risks involved.)

□ Risk Assessment. (cont'd)

- Analyze them.
  - By identifying them after identification, you can understand:
    - when, why and where they might occur.
  - Many techniques to enhance understanding
    - system dynamics models
    - cost models
    - performance models
    - network analysis
    - decision analysis
    - quality risk factor analysis
  - Prioritize them.
    - enables limited resources to be given the most threatening one.
    - by Risk Exposure

□ Risk Control

control instead of *eliminate*
⇓ i.e.,
minimize the risk

- Risk reduction
  - avoiding the risk
    - by changing requirements
  - transferring
    - by allocating risks to other systems
    - by buying insurance to cover financial loss
  - assuming the risk
    - by accepting it and controlling it with the project's resources.

□ The cost of reducing the risk.

$$\{\text{risk leverage}\} =$$

$$\frac{\{\text{risk exposure before reduction }\} - \{\text{the one after reduction}\}}{\{\text{cost of risk reduction}\}}$$

- If the value is not high enough to justify the action,

$\Downarrow$ then

use other less costly or more effective reduction techniques.

□ Summary: Risk Management

- Risk Management
  - Risk Assesment
    - Risk Identification
      - Check List
      - Decomposition
      - Assumption Analysis
      - Dicision Analysis
    - Risk Analysis
      - System Dynamics
      - Performance Models
      - Cost Models
      - Network Analysis
      - Quality Risk Factor Analysis
    - Risk Prioritization
      - Risk Exposure
      - Risk Reduction

- Risk Management (cont'd)
  - Risk Control
    - Risk Reduction
      - Buying Information
      - Risk Avoidance
      - Risk Transfer
      - Risk Reduction Leverage
      - Development Process
    - Risk Management Planning
      - Risk Element Planning
      - Risk Plan Integlation
    - Risk Resolutions
      - Risk Mitigation
      - Risk Monitering and Reporting
      - Risk Reassessment

Chapter 3

Capturing the Requirements

- Chap. 2 Planning (How we plan a project?)
  (Preparation)
  - How long will it take?
  - How much will it cost?
  - What resorece are needed?

- Chap. 3 Capturing Requirement
  (We now start)
  - How to capture? (Elicitation)
  - How to document?
  - How to review? (Verification, Validation)

□ Capturing the Requirements

Requirement $\Leftarrow$ subscribed by customer to developer.

Captureing it $\Rightarrow$ Understanding of what the customers
and users expect the system to do.

□ Contents

- Two types of requirement.
  - Functional & Non-functional
- How to document them.
  - Static descriptions & Dynamic ones
- How to review them for
  - Correctness, Completness and Consistency.
- Requirements specification.

□ Requirenent

$\Downarrow$ is

a description of something the system is capable of doing in order to fulfil the system's purpose.

□ A survey by Standish group[6]

(over 350 companies about their over 8000 software projects.)

- 31% projects were canceled before they were copmleted.

Why?

1. imcomplete requirements (13.1%)
2. lack of user involvement (12.4%)
3. lack of resorces (10.6%)
4. unrealistic expectations (9.9%)
5. lack of executive support (9.3%)
6. changing requirements and specifications(8.7%)
7. lack of planning (8.1%)
8. system no longer needed (7.5%)

[6] The Standish Group: The CHAOS Report (1994); The Scope of Software Engineering. Software Engineering 2(2), pp.126–137.

□ A survey by Standish group Cont'd

To find and fix a requirement problem, it costs:

- $ 1 $\Leftarrow$ during the requirements definition process;
- $ 5 $\Leftarrow$ during design;
- $ 10 $\Leftarrow$ during coding;
- $ 20 $\Leftarrow$ during unit testing;
- $ 200 $\Leftarrow$ after delivery.

□ Process of determinging the requirements.

1. Elicitation
   - By asking questions;
   - By demonstrating similar systems;
   - By developing prototypes of all or part of the proposed system.
2. Documentation
   - So that developers and customers agree on what the system should do.
   - Rewritten in a more mathematical way (Formal Language), so that the designers can transform requirements into a good system design.
3. Verification
   - ensures that the requirements are complete, correct and consistent.
4. Validation
   - makes sure that we have described what the customer intends to see in the final product.

□ Documentation:

- Internal documentation
  - ⋆ At a level appropriate for a programmer
- External documentation
  - ⋆ Intended to be read also by those who may never look at the actual code.

□ Elicitation:

- A variety of techniques exists to determin what the users and customers really want and need.

- Categories of requirements
  (1) Absolutely must be met.
  (2) Highly desirable but not necessary.
  (3) Possible but could be eliminated.
    * if the system will cost too much or take too long to develop, (3) can be dropped, and (2) can be analyzed for elimination and postponement.

□ Requirements specify not "how" but "what."

Requirements do not specify how the system is to be implemented.

$\Downarrow$ i.e.,

A requirement addresses the purpose of the system without regard for how the system is to be implemented.

$\Downarrow$

Some system characteristics having nothing to do with the purpose of the system such as

- Implementation-specific descriptions.
  - What database management system to use?
  - How much memory the computer is to have?
  - What programming language must be used to develop the system?

should be deleted from any requirements specification

$\Downarrow$

Requirements identify the what of the system, while the design identifies the how.

□ Two kinds of requirements document: Definition & Specification

- Requirements definition
  - A listing of everything the customer expects the system to do.
  - Written in terms that the customer can understand.

    $\Downarrow$

    An understanding (between customer and developer) of what the customer needs or wants.

- Requirements Specification
  - Restates the requirement definition in technical terms for the development and design.
  - Should be written by *requirements analysts* and designers.

Note that:
Configuration Management Methods which are used throughout the development begin here by linking the definition and specification.

□ Configuration Management Methods:

- A set of procedures that track:
  - the requirements that define what the system should do;
  - the design modules that are generated from the requirements;
  - the program code that implements the design;
  - the tests that verify the functionality of the sysem;
  - the documents that describe the system.

□ To make requirment testable:

- Robertson's proposal:
  - Do not use adverb and adjective for a quantitative description.

    e.g., immediately ⇒ 5 miniutes

  - Do not use pronouns.

    e.g., they ⇒ users

  - Make sure that every noun is defined in exactly one place.

□ Functional/Non-functional Requirement:

- Functional requirements
  - describes an interaction between the system and its environments.

  e.g.,
    - ∗ What states are acceptable for the system to be in?
    - ∗ How the system behaves given stimmuli?

- Non-functional requirements
  - describes an *restrictions* on the system that limits our choices for a solution.

□ Three purposes of requirement:

- allow **developers** to explain their understanding of how the customer wants the system to work;
- tell **designers** what functionality and characteristics the system is to have;
- tell the **test team** what to demonstrate to convince the customer that the system being delivered is what was ordered.

☐ Assume now that you are a customer ordering:

- Bus booking system.

  or

- A convenience store system.

What are your requirements?

□ Toy example of requirements for a bus booking system.

- Input from a terminal should be date, time destination, place of departure and male/female to the master computer from any of 20 terminals
- Which sheets are free for the request?
- Male should not be seated next to female.
- The final answer should be your seat is #... or no seat.
- etc

□ Check List:

- Are the requirements
  - correct? (without errors?)
  - consistent?
    * inconsistent if it is impossible to fulfil multiple requirements simultaneously
  - complete?
    * Are all possible states, state changes, inputs, described?
  - realistic?
  - verifiable?
  - tracable?
    * The system shall provide real-time response to queries.

      $\Downarrow$

      ... respond to queries in not more than 2 seconds.
    * In identifying the position of the satelite, position should be fairy precise.

      $\Downarrow$

      ... position error shall be less than 50 feet along orbit, less than 30 feet off orbit.
  - Each one really needed by customers?

$\Downarrow$

Then how to express them?

□ How to express Requirements:

For many years,
requirements were specified in the customer's natural language.

$\Downarrow$

imprecise and unambiguous

e.g., "*available*"
for customers:

- The system is "*unavailable*" to the user, when users are unable to use a terminal because system backup is in progress.

* for developers:

- The system is still functional and hence "*available*".

$\Downarrow$

Software engineers have investigated many ways to define requirements in a more rigorous way.

$\Downarrow$

Formal notations. (e.g. in z.)

□ Static & Dynamic description:

A system description lists:

· the system entities or objects;

· their attributes;
(including the functions that can be performed to them or by them);

· their relationships with each other.

$\Downarrow$

- Relationship among its entities.

$\Downarrow$

Difficult to explain how the system reacts over a period of time.

$\Downarrow$ That's why it's called

static

$\Downarrow$

When time is not important in the system's operation,

$\Downarrow$

it is usuful and adequate.

□ Static & Dynamic description: cont'd

- Description of a system in terms of states and stimuli.

$\Downarrow$

dynamic

$\Downarrow$

The system is considered to be in a particular state until some stimulus causes it to change its state.

starting by describing all possible states and stimuli.

□ Formal System Descriptions:

1. Static Descriptions

- Recurrence Relations
  - Initial condition is defined.
  - The transformation rule from one condition to the next is described.

- Axiomatic Definition
  - Basic system properties are specified ⇒ **axioms**.
  - The behavior of the system ⇒ **theorems**. (in the sense that the behaviour generates new properties.)

- Expression as a language
  - Usually by Backus-Naur form.

- etc.

Note that: They are usufull only when time is not important.

□ Formal System Descriptions: cont'd

2. Dynamic Description

... How to describe a system in terms of states and stimuli.

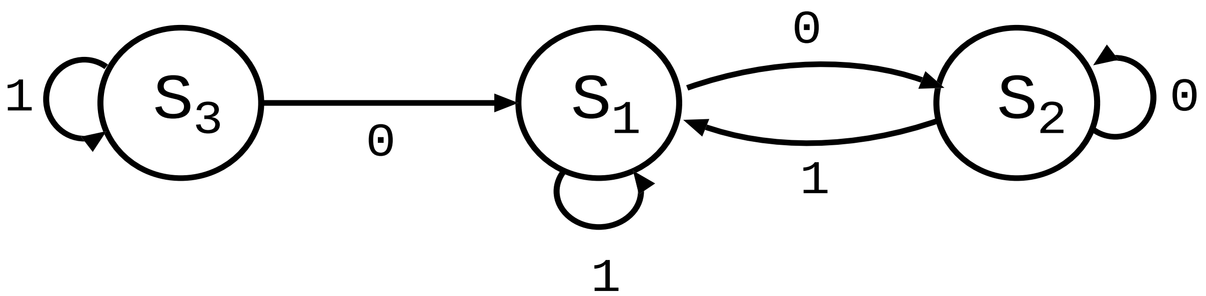
- Decision Tables
  - Describe a system as a set of
    * possible conditions satisfied by the stsatem at a given time;
    * rules for reacting to the stimuli when certain sets of the conditions are met;
    * actions to be taken.

- Transition Diagrams.
  - Draw a circle for each state in the system;
  - For each state and possible imput, draw a directed arrow to indicate the state transition. (in the sense that the behaviour generates new properties.)

□ Dicision table:

	rule-1	rule-2
Condition-1	T	F
Condition-2	F	F
Condition-3	–	T
action-1		
action-2		

□ Transition diagram and table:



	0	1
S_1	S_2	S_1
S_2		
S_3		

2. Dynamic Description cont'd

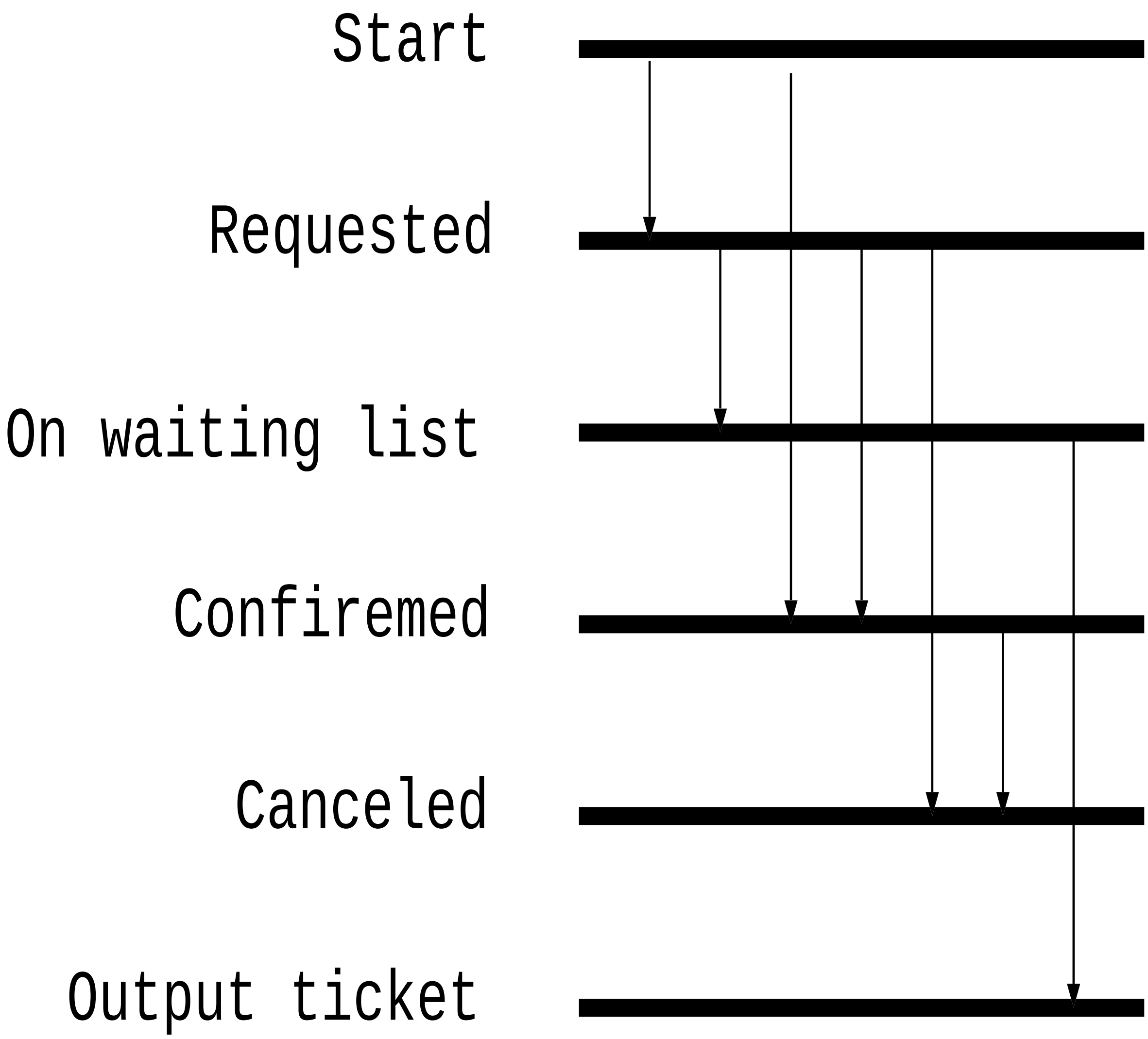
- Fence Diagram

- UML (Unified Modeling Language)

- Event Tables.
  - raw: states
  - column: events
  - cell: action/no-action/not-possible
    no-action: when an event that was for another state was occured.)

- Petrinet

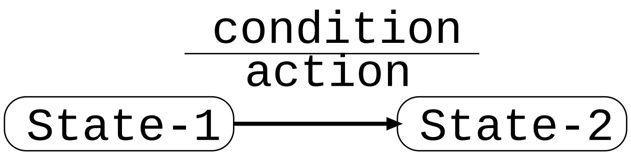
Fence Diagram:



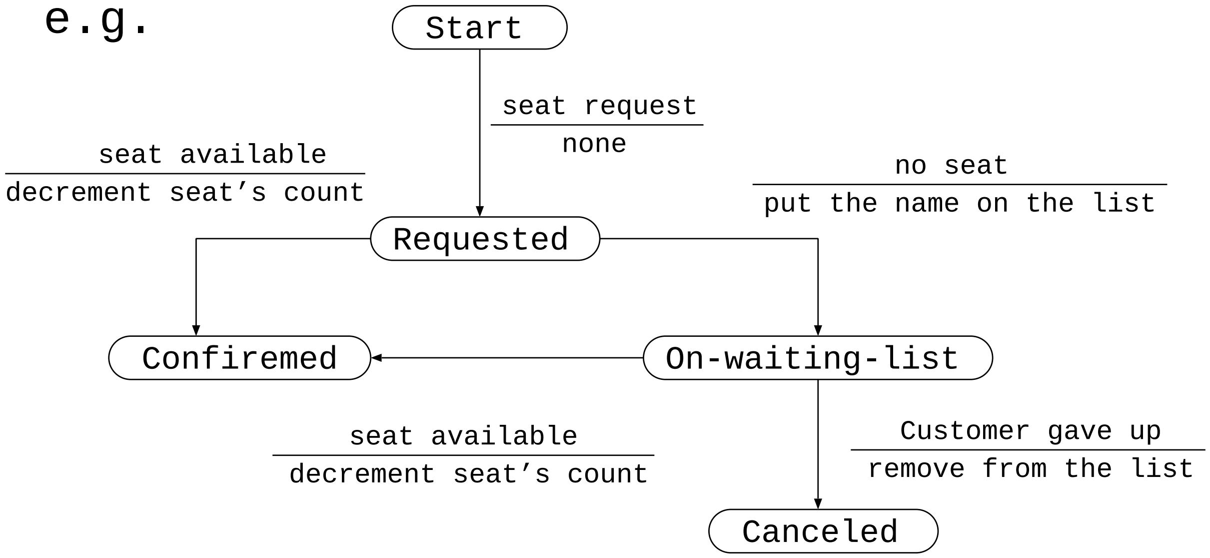
□ Event Table:

	Event-1	Event-2	Event-3
State-1	action 1	no action	
State-2			
State-3	not possible		

UML (Unified Modeling Language)



e.g.



Decision Tables, Transition Diagrams Event Tables

$\Downarrow$

only useful for systems whose states and events occur in sequence.

$\Downarrow$

How about, when several events occur at once?

$\Downarrow$

The systm must perform parallel processing.

$\Downarrow$

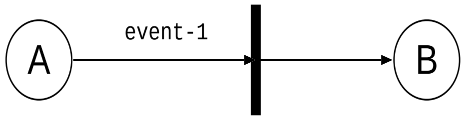
We need to **synchronize** events.

$\Downarrow$

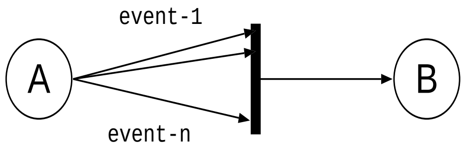
Description by **Petri Nets.**

□ Petri Net:

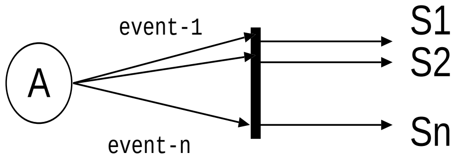
f(state-A, event-1) = state-B



f(state-A, E-1, E-2, ..., E-n) = state-B

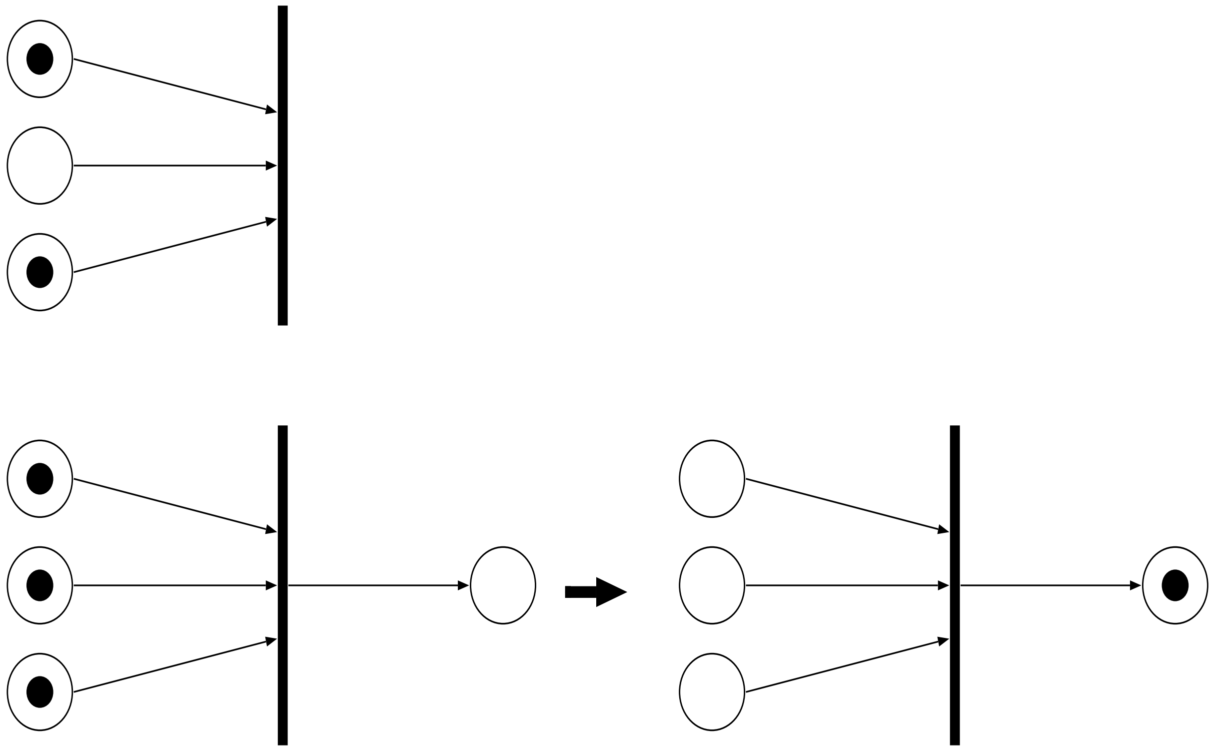


f(A, E-1, E-2, ..., E-n) = S1, S2, ... ,Sn



□ Synchronization in Petri Net:

- Events that occur are represented by tokens.
- Transitions are described by firing rules.



Note: No one approach is universally applicable to all systems.

- Some are better at capturing

  non-functional requirements than others ⇒

  it may be best to use

  – one approach to record the functional ones;
  – another to describe the constraints.

- Some are good at describing deta requirement,

  – one approach for data;
  – another to process or time-related activities

Chapter 4

Designing the System

- So far we have studied
  - Planning;
  - Capturing requirement.

  Now we are going into
  - Designing.

- In this chapter, we look at
  - *conceptual* and *tecnical* Design
  - styles, and techniques;
  - what is good design;
  - validating designs;
  - documenting the design.

□ Designers move back and forth among activities like:

- understanding the requirements;
- proposing possible solutions;
- testing aspects of a solution for feasibility;
- presenting possibilities to the customers;
- documenting the design for the programmer.

□ Conceptual & Technical Desings:

- Conceptual Design:
  - What the system is to do.

    $\Rightarrow$ for customer

- Technical Design:
  - How the system is to work.

    $\Rightarrow$ for programmer

□ Conceptual & Technical Desings: cont'd

- A good conceptual design:
  - should be written in the customer's language;
  - should contain no technical jargon;
  - describes the functions of the system;
  - is independent of implementation.

- Technical desigh includes:
  - a description of the hardware components and their functions;
  - the hierarchy and function of the software components;
  - data structures and their flow.

□ Decomposition & Modularity:

To design a system is to determine a set of components and intercomponent interfaces.

$\Downarrow$

decomposition of the design into modules.

$\Downarrow$

A system is called *modular* when each activity of the system is performed by exactly one component, and when the inputs and outputs of each componet are well-defined.

$\Downarrow$ How we separate the desing?

- Five ways suggested by Wasserman:
  - Modular decomposition
  - Data-oriented decomposition
  - Event-oriented decompositon
  - Outside-in design
  - Object-oriented design

□ Decomposition & Modularity: cont'd

Decomposition starts with a high-level depiction of the system's key elements,

$\Downarrow$then

creates lower-level looks at how the system's features and functions will fit together.

$\Downarrow$e.g.,

- Modular decomposition
  - based on
    - assigning functions to components
  - high-level description of
    - functions to be implemented
  - lower-level description of
    - how each component will be organized and related to others.

□ Decomposition & Modularity: cont'd

- Data-oriented decomposition
  - based on
    - ∗ external data structures.
  - high-level description
    - ∗ dipicts general data structures.
  - lower-level descriptions provide
    - ∗ detail on what data elements will be involved, and
    - ∗ how the are related.
- Event-oriented decompositon
  - based on
    - ∗ events that the system must handle, and

    uses
    - ∗ information about how events change the system's state.
  - high-level description catalogs
    - ∗ the various states
  - lower-level descriptions describe
    - ∗ how state transformation take place.

□ Decomposition & Modularity: cont'd

- Outside-in design
  - based on
    - user inputs to the system.
  - high-level description lists
    - all possible inputs.
  - lower-level descriptions address
    - what the system does with each input and what outputs are produced.
- Object-oriented design
  - identifies classes of objects and ther interrelationships.
    - assigning functions to components
  - at the highest-level
    - each object type is described.
  - at lower-levels
    - the object attributes and actions are discussed.

□ Top-down Design or Bottom-up?

Three design levels:

- Architecture design:
  - designs
    - system capabilities.
  - described by modules and their interconnections.
- Code design
  - involves
    - algorithms and data structures.
  - described by programming language's primitives.
- Executable design
  - involves
    - the code design at a lower level of details such as
      - memory allocations
      - data formats
      - bit patterns

Back and forth in reality.

Architecture level:

```
Input from a terminal
Send it to a central computer
    ......
Output answers
```

Code design level:

```
main(){
  verify_inserted_card();
  ask_passwd();
      ......
  }
```

Executable design level:

```
#def N  5,000,000,000
  ......

int customer[N][M];
  ......

verify_inserted_card(){
  ......
}
  ......

main(){
  verify_inserted_card();
  ask_passwd();
      ......
  }
```

□ Modularity and Levels of Abstraction:

In a modular design, each component has well defined inputs and outputs, and a clearly stated purpose;

$\Downarrow$ furthermore

modular components are organized in a hierarchy as the result of decomposition.

$\Downarrow$

By combining modular components with several levels of abstruction, we get several different views of the system.

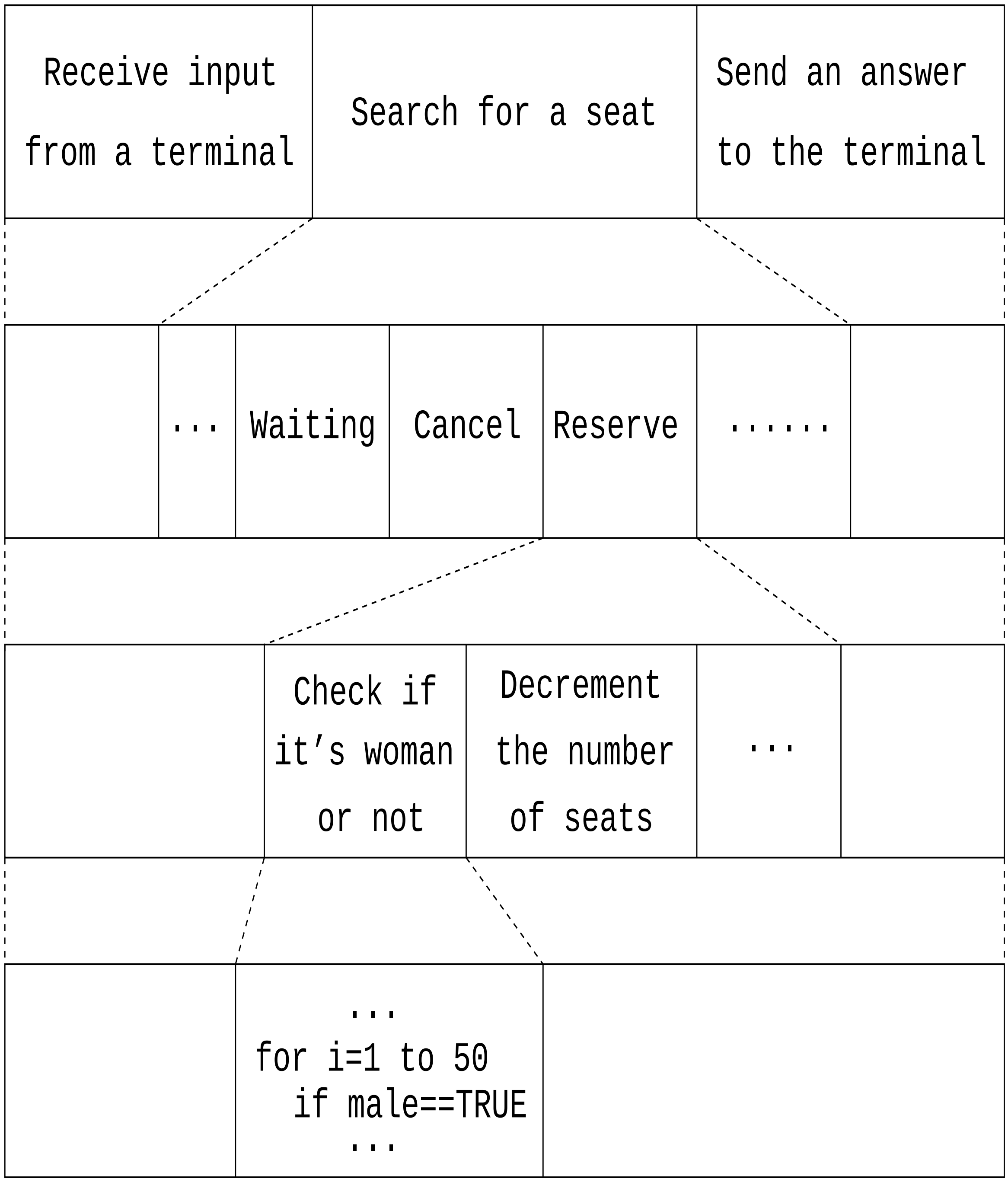
$\Downarrow$ So,

we can investigate the system one level at a time.

$\Downarrow$

As we need more detail about a portion of the system, we move to lower levels of abstraction.

A long-distance bus seat booking system:



□ **Modularity and Levels of Abstreaction:** cont'd

Bonus:

$\Downarrow$

We can design differnet components in different ways.

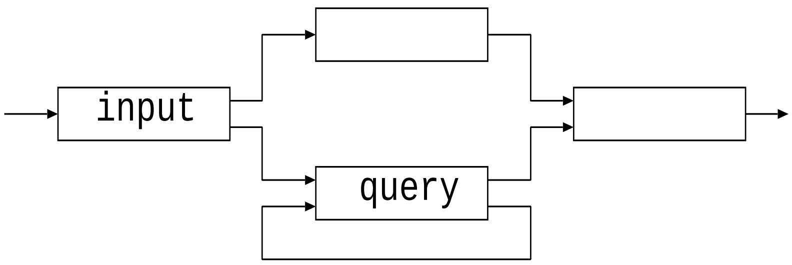
$\Downarrow$ e.g.

- the user interface with object-oriented,
- security part with state-transition diagram,
- etc

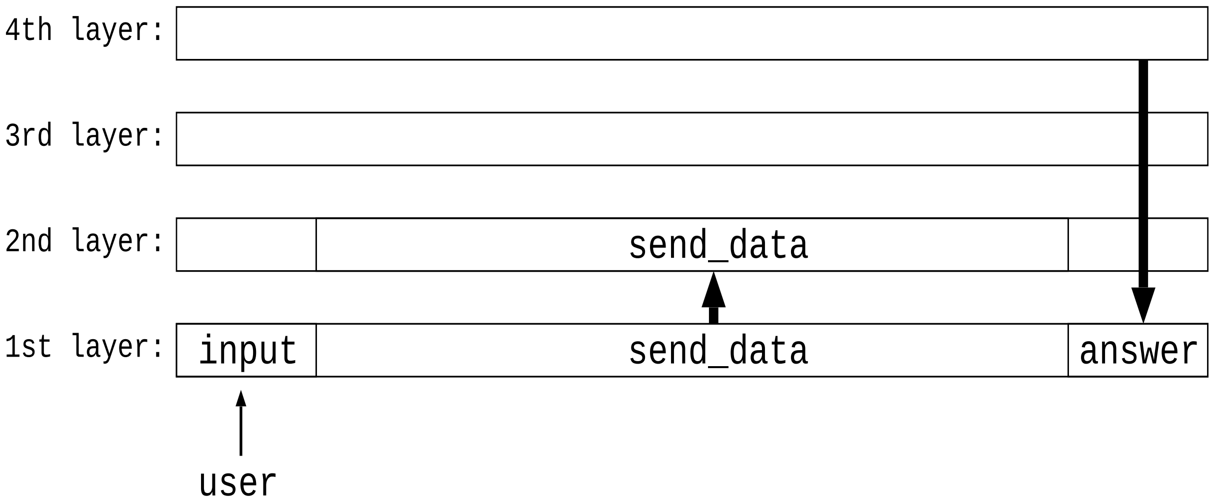
□ Design styles:

- Pipes and Filters
  - Filters can be re-used easily.
  - It is easy to add new filters and remove old ones.
  - Concurrent execution is allowed.
- Layering
  - Layers are hierarchical:
    - Each layer provides services to the one next higher.
  - Each layer has access to some/all of the other layers or only to adjacent layer.
- Client-server
  - Under a distributed system.
  - a *client* requests an service (action) and a *server* responds to the request.
- etc

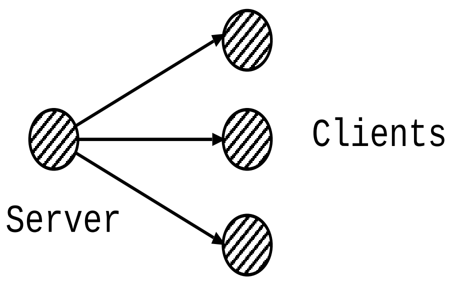
pipes and filters:



Layering:



Server & Client:



□ Layered Architecture:

- Easy to implement abstraction
  - each layer ⇒ increasing level of abstraction.
- Easy to modify/add/remove.
  - Because a layer usually affects only two adjacent layers
- Easy to reuse a layer.

- But note that it is not always easy to structure a systme in layers.

□ Home work:

1. Choose one of the following toy projects.
   (1) A long-distance bus seat booking system.
   (2) ATM system for a bank.
   (3) A system for a convenience store.
2. Assume that you are a customer and submit requirement in daily English for the system that you chose above.
3. Describe a *requirment definition* of the requirment submitted above.
4. When you describe a *requirments specification* for the above, which do you use (1) static or (2) dynamic way and what technique in it will you pick up for that.?
5. Describe it by using the technique that you chose above.
6. Decompose the problem and describe both *conceptual* and *technical design*
5. Which style do you use for which module?
6. Discuss "concurrency" for the poblem you chose.
7. Discuss about fault,failure and error for the poblem you chose.

□ Concurrency:

How to maintain consistency of the data shared among components that execute concurrently.

1. **By carefull timing.**
   - One way to handle concurrency by prescribing the amount of time allocated to the performance of any action.

     $\Downarrow$ Careful *timing* can ensure that
   - one action does not interfere with another.

     $\Downarrow$ However, what happens
   - if the system should react to external events like the sensor data change.

     $\Downarrow$
2. **By synchronization**

□ **Concurrency:** cont'd

2. Syncronization of concurrent processes.
   - **Mutual exclusion**
     * When one process is accessing a data elements, no other process can affect that element.

       or
     * If two operations can affect the state of *a shared object*, then they sould be executed using this scheme,

       e.g.,
     * if an operation tests the state of an object, then the object should be locked so that the state does not change during the test is done and action is taken

   - **Process/component priority**
     * The one with higher priority can win the battle between the two, locking out the other until the higher-priority one is completed.

□ Concurrency: cont'd

We want to design systems so that they are always correct independent of the timing.

$\Downarrow$ Two methods for the purpose.

- **Monitors**
  - A monitor is an abstract component that controls the mutual exclusion of a particular process.
    - **M** $\Rightarrow$ the monitor of a process **P**
    - **A, B** $\Rightarrow$ concurrent two components.

    - When component **A** wishes to execute **P**, it must consult **M** to obtain access to **P**.
    - If component **B** wishes to execute, as **A** is executing **P**, then **M** suspends **B** until **A** finished with **P** and afterwords **M** activates **B** to allow its access to **P**.
  - The monitor has usually a condition cheker to make sure that conditions are satisfied for invoking the requested process.
- **Gardians**

□ Concurrency: cont'd

- **Gardians**
  - A task that is always running with its only purpose being to control access to a resorce (a data-sturcture or a process)
    - **G** $\Rightarrow$ a guardian of a resource **R**.
    - **T** $\Rightarrow$ a task.

    - **T** must issue a request to **G** to use **R**
      (Interaction of T and G is called a *rendezvous.*)
    - **G** suspends **T** until the condition checker says that the condition is fulfilled.

      $\Downarrow$ Thus
    - **G** always reacts to a valid task.

□ What is good design:

- Component Independence.
- Exception Identification and Handling.
- Fault Prevention and Fault Tolerance.
- Psychological Effect

□ Component independence:

- six levels of coupling
  - content coupling
  - common coupling
  - control coupling
  - stamp coupling
  - data coupling
  - uncoupling
- seven levels of cohesion
  - coincidentally cohesive component
  - logically cohesive component
  - temporally cohesive component
  - procedurally cohesive component
  - communicationally cohesive component
  - sequential cohesive component
  - functional cohesive component

□ Component Independence:

The degree of component independence in terms of:

- **coupling**
  - **content coupling**
    - When one component modifies an internal data in another component.
    - When one component branch into the middle of another component.
  - **common coupling**
    - When it is difficult to know which component is responsible for having changed a particular variable.

      $\Downarrow$ e.g.,

      · component-1 $\Rightarrow v = 0$;

      · component-2 $\Rightarrow$ increment$v$;

      · component-3 $\Rightarrow v = u + w$;
  - **control coupling**
    - When one component passes parameters to another component to control the activity of the component.
  - **stamp coupling**
    - When a data-structure is used to pass information from one component to another.
    - When the the data-structure itself is passed.

- **data coupling**
  * When only data instead of data-structures are passed. (If coupling should exist this is the most deirable.)
- **uncoupling**
  * When components have no interconnections at all (Completely independent.)

- **cohesion**

  the degree of *cohesiveness* among the internal parts of a component.

  $\Downarrow$

  "glue" a component is constructed with.

  $\Downarrow$ So our goal is

  to make each component as cohesive as possible

  $\Downarrow$ so that

  every part of a component's processing was related to the component's singular function.

  - **conincidentally cohesive component**
    * The worst degree of cohesion
    * Parts in a component are un-related to one another.

      $\Downarrow$ i.e.
    * unrelated functions, processes or data are in the same component for reasons of convenience.

– **logically cohesive component**
  * The next higher level of cohesion (still not desirable.)
  * several logically related functions or data-elements are in the same component.

    $\Downarrow$ e.g.

  * all kinds of inputs regardless of where they are from or how they are used

    $\Downarrow$

  * differnet pupose for different components

    $\Downarrow$

  * many unrelated functions in one place

– **temporally cohesive component**
  * When a component performs several functions in sequence but the functions are related only by the *timing*.

When components are logically or temporally cohesive, it is difficult to modify a system function X. Since logically or temporally cohesive components performs several different functions, all components that include the part related to the function should be searched.

– **procedurally cohesive component**
  * When several functions are together in a component just to ensure the order to be performed,

    $\Downarrow$ e.g.,

    data must be entered before they can be checked and then manipulated.

- **communicationally cohesive component**
  * When several functions are together in a component because they produce or operate on the same data set,

    $\Downarrow$ e.g.,

    unrelated data are fetched together because they are in a disk.
- **sequential cohesive component**
  * When the output from one part of a component is input of the next part.
  * The component may still not constrain all of the processing that relats to a function.
- **functional cohesive component**
  * Every processing element is essential to the perfromance of a single function and all essential elements are contained in one component.

    $\Downarrow$

    Our ideal.

□ Fault & Failure: revisited

- Not every *fault* corresponds to a *failure*.
- Developers see *faults* while customers see *failures*.
- Correct *fault* before it becomes a *failure*.

□ A good design should

- detect fault;
- correct fault;
- tolerate fault.

□ Fault Detection:

- Anticipate when failures will occur
  ⇒ **active fault detection,**

rather than

- waiting *failures* to occur
  ⇒ **passive fault detection.**

□ Fault Detection: cont'd

- Use of redundancy
  - results of two different processes for the same purpose are compared to determine if they are the same
  - If two funtionally equivalent systems are designed by two different design teams at two differnet times using diffent techniques, the chance of the same fault occuring simultaneously is very small.

$\Downarrow$ further

n-version programming

$\Downarrow$ e.g.,

In the system for the space shuttle,

seven computers vote to determine the next operation.

□ Fault Correction:

fixes the damage done by the fault and changes the product to eliminate the fault.

$\Downarrow$ So,

a good design should include a strategy to handle faults as they are found,

$\Downarrow$ e.g.

Stop the system when a failure occurs,

or

record the failure and keep the state of the system at the moment of the failure so that we can return to fix it later.

□ Fault Tolerance:

Usually, correcting a fault is too expensive, risky and incovenient.

$\Downarrow$ So,

Design minimizes the damage instead of a correction.

$\Downarrow$ e.g.

A banking system may switch to a backup processor or make duplicate copies of data and transactions.

$\Downarrow$ But

□ Fault Tolerance: cont'd

It is said that there is no faultless system in the world.

$\Downarrow$ That is,

the more complex systems are, the more likely to have significant faults.

$\Downarrow$ And,

the code to implement fault tolerance may itself contain failures which may cause irreparable damage.

$\Downarrow$

Are there any techniques that designs fault-tolerant without adding risk?

$\Downarrow$ e.g.

Fault tree analysis
Failure-mode anlysis

□ Fault Tree Analysis:

Trees that display the logical path from effect to cause.

1. Identify possible failures.

   e.g.

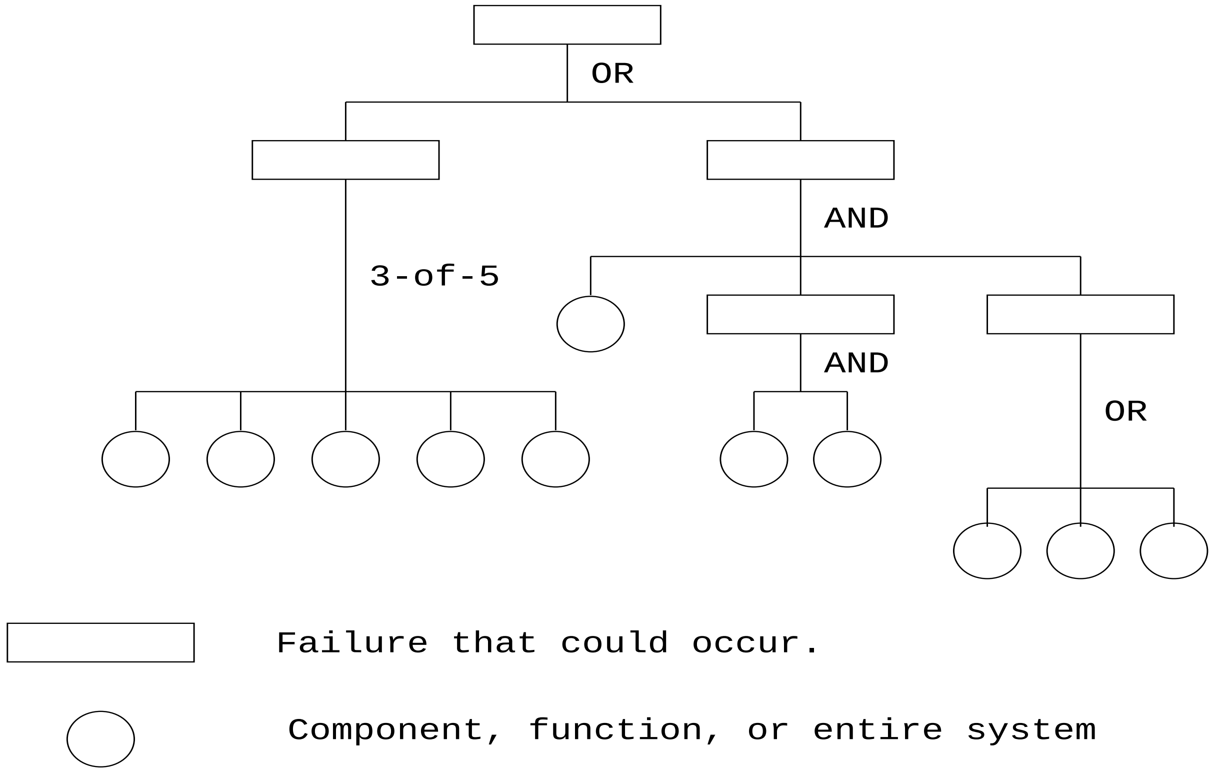
   – by using a set of guidewords to know how the system might deviate from its intended behavior, such as:
     * what if
       · no data or control signal was sent or received?
       · the volume of the data is too much or the flow is too fast?
       · the volume of the data is too low or the flow is too slow?
       · the data or control signal is incomplete?
       · the data or control signal has another component?
       · the signal arrives too early for the clock?
       · the signal arrives too late for the clock?
       · the signal arrives too early in the expected sequence?
       · the signal arrives too late in the expected sequence?

□ Fault Tree Analysis: cont'd

2. Build a graph
   - whose nodes are
     - failures;
     - single component;
     - system function; or
     - entire system.
   - and edges indicate
     - the relationships among nodes, i.e.,
       * AND
       * OR
       * n-of-m
         · n components' failure out of m redundant components lead to a system failure.

3. Search for several types of design weaknes such as
   - *single points of failure.*

An example of fault-tree



□ Design Validation & Verification:

- Validation
  - Does the design satisfy all requirements specified by the cusotmer?
- Verification
  - Does the quality of design is high enough?

□ Mathematical Validation:

1. Break the system into a set of processes,
   so that each process has
   - a set of inputs
   - a set of expected outputs
   - a set of assetions about the processes.
2. For each process, demonstrate:
   (1) if the set of inputs is formulated correctly, it is transformed properly into the set of expected outputs.
   (2) the process terminates without failure.

$\Downarrow$

This proves that the design is correct.

$\Downarrow$

time-consuming & expensive

$\Downarrow$

less formal methods

$\Downarrow$

□ Card & Glass's Method:

□ Card & Glass's Method:

- With the goal of seaking a good predictor of *faults*, they discuss:
  - the distribution of faults does not depend on the type of *coupling* in design.
  - components with larger spans of control tended to have a higher fault rate;
  - more cohesive modules had lower fault rates.
- They pointed out that:
  - design complexity involves:
    * complexity within each component
    * complexity of their relationships

□ Card & Glass's Method: cont'd

- They defined a mesure of system complexity as:

$$C = S + D.$$

Here,

- · S is called the structural (between-component) complexity,
- · D is called the data (within-component) complexity.

and are given as:

$$S = \frac{\Sigma\, f^2(i)}{n}, \qquad D = \frac{V(i)}{f(i)+1},$$

where

- · $f(i)$ is the fan-out of component i,
- · $V(i)$ is the number of input and output variables in component i
- · n is the number of components.

Chapter 5

Writing the Programs

Most software is developed by teams.

$\Downarrow$

Cooperation and coordination is required.

$\Downarrow$

So, it is important for others to be able to understand:

- what you have written?
- why you have written?
- how it fits in with their work?

□ What are said to be important:

- Standardization;
  - · Throw your own method away. Standardize!
- Importance of comments;
  - · header comment
  - · other program comment

    e.g.,
    cases where the denominator might be zero
    $\Downarrow$ should be pointed out
    when this might occur and how it is handled.

- meaningful variable name
  - · $i = i + 1$; /* increment i */
  - · $i = i + 1$; /* set counter to read next case */
  - · case_counter = case_counter + 1;

- Start by pseudocode;

- Revising and rewriting not patching;

  - · Find an essential solution instead of tentative one.

- Revision control.

□ The Revision Control System (RCS):

- manages multiple revisions of files;
- automates the storing, retrieval, logging, identification, and merging of revisions;
- is useful for text that is revised frequently, or that is revised by multiple person.

□ The basic user interface::

- **ci** (check in)
  - deposits the contents of a file into an archival file called an RCS file which contains all revisions of a particular file.
- **co** (check out)
  - retrieves revisions from an RCS file.

□ How to use RCS::

Suppose you have a file f.c that you wish to put under control of RCS.

1. Make a directory named RCS by:

   % mkdir RCS.

2. Then invoke the check-in command

   % ci f.c

   * This command creates an RCS file in the RCS directory, stores f.c into it, and deletes f.c
   * If it is the first version, it asks you for a description.

     enter description, terminated with single '.'
     NOTE: This is NOT the log message!
     >>

   * The description should be a synopsis of the contents of the file.
   * All later check-in commands will ask you for a log entry, which should summarize the changes that you made.

3. Files in the RCS directory are called RCS files; the others are called working files. To get back the working file f.c in the previous example, use the check-out command

   % co f.c

   * This command extracts the latest revision from the RCS file and writes it into f.c.
   * If you want to edit f.c, you must lock it as you check it out with the command

   % co -l f.c

4. You can now edit f.c

5. Suppose after some editing you want to know what changes that you have made. The command

   % rcsdiff f.c

   * tells you the difference between the most recently checked-in version and the working file.

6. You can check the file back in by invoking

   % ci f.c

   * This increments the revision number properly.

7. If ci complains with the message

   ci erro: no lock set by (your-name),

   then you have tried to check in a file even though you did not lock it when you checked it out. Of course, it is too late now to do the check-out with locking, because another check-out would overwrite your modifications. Instead, invoke

   % rcs -l f.c

   * This command will lock the latest revision for you, unless somebody else got ahead of you already.

   Locking assures that you, and only you, can check in the next update.

   Even if a revision is locked, it can still be checked out for reading, compiling, etc.

   If your RCS file is private, i.e., if you are the only person who is going to deposit revisions into it, strict locking is not needed and you can turn it off. If strict locking is turned off, the owner of the RCS file need not have a lock for check-in; all others still do. Turning strict locking off and on is done with the commands

   % rcs -U f.c

   % rcs -L f.c

- To avoid the deletion of the working file during check-in (in case you want to continue editing or compiling), invoke

  % ci -l f.c, or

  % ci -u f.c

  * These commands check in f.c as usual, but perform an implicit check-out.
  * The first form also locks the checked in revision,
  * the second one doesn't.

  Thus, these options save you one check-out operation. The first form is useful if you want to continue editing, the second one if you just want to read the file. Both update the identification markers in your working file.

  You can give ci the number you want assigned to a checked in revision. Assume all your revisions were numbered 1.1, 1.2, 1.3, etc., and you would like to start release 2.

- The command

  % ci -r2 f.c, or

  % ci -r2.1 f.c, etc.

  assigns the number 2.1 to the new revision.

- The corresponding co commands

  % co -r2 f.c, and

  % co -r2.1 f.c

  retrieve the latest revision numbered 2.x and the revision 2.1, respectively. (co without a revision number selects the latest revision)
  For example, to start a branch at revision 1.3, invoke

  % ci -r1.3.1 f.c

  This command starts a branch numbered 1 at revision 1.3, and assigns the number 1.3.1.1 to the new revision.

- RCS can put special strings for identification into your source and object code. To obtain such identification, place the marker

  Id

  into your text, for instance inside a comment. RCS will replace this marker with a string of the form

  $Id: (filename) (revision) (date) (time) (author) (state) $

- With such a marker on the first page of each module, you can always see with which revision you are working. RCS keeps the markers up to date automatically.

- You may also find it useful to put the marker
  - Log

  into your text, inside a comment. This marker accumulates the log messages that are requested during check-in. Thus, you can maintain the complete history of your file directly inside it.

Thus RCS:

- Store and retrieve multiple revisions of text.
  - RCS saves all old revisions in a space efficient way. Changes no longer destroy the original, because the previous revisions remain accessible. Revisions can be retrieved according to ranges of revision numbers, symbolic names, dates, authors, and states.
- Maintain a complete history of changes.
  - RCS logs all changes automatically. Besides the text of each revision, RCS stores the author, the date and time of check-in, and a log message summarizing the change. The logging makes it easy to find out what happened to a module, without having to compare source listings or having to track down colleagues.
- Resolve access conflicts.
  - When two or more programmers wish to modify the same revision, RCS alerts the programmers and prevents one modification from corrupting the other.
- Maintain a tree of revisions.
  - RCS can maintain separate lines of development for each module. It stores a tree structure that represents the ancestral relationships among revisions.
- Merge revisions and resolve conflicts.
  - Two separate lines of development of a module can be coalesced by merging. If the revisions to be merged

affect the same sections of code, RCS alerts the user about the overlapping changes.

- Control releases and configurations.
  - Revisions can be assigned symbolic names and marked as released, stable, experimental, etc. With these facilities, configurations of modules can be described simply and directly.
- Automatically identify each revision with name, revision number, creation time, author, etc.
  - The identification is like a stamp that can be embedded at an appropriate place in the text of a revision. The identification makes it simple to determine which revisions of which modules make up a given configuration.
- Minimize secondary storage.
  - RCS needs little extra space for the revisions (only the differences). If intermediate revisions are deleted, the corresponding deltas are compressed accordingly.

Chapter 6

Testing the Program

Are tests in order to show a program performs properly?

$\Downarrow$ No!

Test a program to show the existence of a *fault*.

□ Again again, Fault & Failure:

The failure may be the result of any of the following faults:

- The specificatin may be wrong or have a missing requirement.
- The specification may contain a requirement that is impossible to implement.
- The system design may contain a fault.
- The program design may contain a fault
- The program code may be wrong.

A software fault is called a bug?

$$\Downarrow$$

Did the fault wander into the code from some external source?

□ A Fault of Omission/Commission:

- A fault of omission
  - A fault when some key aspect of the code is missing

    e.g.
    when a variable is not initialized.
- A fault of commision
  - A fault when some key aspect of the code is incorrect,

    e.g.
    when variables are initialized to the wrong value.

□ Types of Faults:

- Algorithmic faults
- Syntax faults
- Computation and precision faults
- Documentation faults
- Stress or overload faults
- Capacity or boundary faults
- Timing or coordination faults
- Throughput or performance faults
- Recovery faults
- Hardware and system software faults
- Standards and procedures faults

- Algorithmic faults
  - It occurs when a component's algorithm or logic does not produce the proper output for a given input.
    - ⋆ forget to initialize variables.
    - ⋆ forget to test for a particular condition such as division by zero.
    - ⋆ compare variables of inappropriate data types.
- Syntax faults
  - Fortunately, compilers cathch many of them nowadays.
    - ⋆ The first US space mission to Venus failed by a missing comma in a Fortran do loop.
- Computation and precision faults
  - It occurs when a formula's implementation is wrong or does not compute the result to the required degree of accuracy.
    - ⋆ combine integer and fixed- or floating-point variables.
    - ⋆ improper use of floating-point data, unexpected truncation cause less-than-acceptable precision.
- Documentation faults
  - When the documentation does not mathch what the program actually dees.
- Stress or overload faults
  - When the data structres are filled over their capacity
- Capacity or boundary faults

  - When the system's performance becomes unacceptable as system activity reaches its specified limit.
- Timing or coordination faults
  - When the code coordinating events is inadequate.
- Throughput or performance faults
  - When the system does not perform at the speed prescribed by the requirement.
- Recovery faults
  - When a failure is encountered and the system does not behave as the designers desire of as the customer requires.
    - ⋆ if a power failure occurs.
- Hardware and system software faults
  - When the supplied hardware and system software do not actually work according to the documented operating conditions and procedues.
- Standards and procedures faults
  - When the organizational standards and procedures have not been followed.

□ Test Organization:

- Unit test
  - verifies that the component functions properly.
- Integration test
  - verifies that the components work together.
- Function test
  - verifies if the functions described by the developer's requirements specification of software and hardware are actually performed.
- Performance test
  - compares the system with the remainder of these software and hardware requirements.
- Acceptance test
  - the system is checked against the customer's requirements description.
- Installation test
  - is run to make sure that the system still functions in the environment in which it will be used.

□ Unit Test:

- Examining the code
- Proving Code Correct
- Testing Program Components
  - Testing vs. Proving
- Comparing Techniques

□ Examining the code:

- Walk-throughs or Inspections?
  - Walk-through
    - code and documentation are presented to the review team and the team comments on their correctness
    - the programmer leads the discussion
    - informal
  - Inspection
    - review team checks the code and documentation against a prepared list of concerns.
    - more formal
    - e.g.
      - 1st meeting for an overview of the code and for a description of the inspection goals.

        $\Downarrow$
      - member prepare for a 2nd meeting noting faults found

        $\Downarrow$

        ...
      - in the meeting members report what they have found recording additional faults in discussing individuals'findings.

□ **Examining the code:** cont'd

Typical task of inspection at Hewlett Pckard (Grady, 1977)

- planning for an inspection **(2 hours)**

$\Downarrow$

- team meeting **(30 mins)**

$\Downarrow$

- individual preparation **(2 hours)**

$\Downarrow$

- recording individual's findings **(90 mins)**

$\Downarrow$

- team brainstorming the findings and recommendaing actions to be taken **(30 mins)**

$\Downarrow$

- write and release a sumary **(30 mins)**

☐ Examining the code: cont'd

Software Engineer Researchers' investigations

- Fargam (1976)
  - 67% of the faults eventually detected were found by inspection.
  - Inspection group had 38% fewer failures during the first 7 months of operation than walk-through.

- Ackerman et al. (1986)
  - 93% of all faults in a 6000 line business application were found by inspection
- Jones(1977)
  - 85% of the total faults found are by inspection.
  - (no other method removed even half)

□ Proving Code Correct:

- A code has been

$$\Downarrow$$

(1) coded by you

$$\Downarrow$$

(2) examined by you

$$\Downarrow$$

(3) reviewed by a team

$$\Downarrow$$

(4) **proving the code**

□ Formal Proof of a Code:

- view the code as a flow of statements (assertion)

  statement ⇒

  ∗ combinations of logical variables
  (each of which is true or false)

  ∗ connected by the logical connectives such as

  · conjunction

  · disjunction

  · negation

  · implication

  · equivalence

  · universal quantifier

  · existential quantifier

□ An Example of a Proof:

$T'(N) = T(N);$
$exchange\_not\_occur = 0;$

while $(exchange\_not\_occur < N)$
  for $(i = 0 \text{ to } N)$
    if $(T'(i) > T'(i+1))$
      exchange $T'(i)$ with $T'(i+1)$;
    else
      $exchange\_not\_occur$ ++;

A_s: (T is an array) **AND** (T is of size N)

A_f: (T' is an array) **AND** ($\forall i$ if $i < N$ then $(T'(i) \leq T'(i+1))$ **AND** ($\forall i$ if $i \leq N$ then $\forall j(T'(i) = T'(j))$ **AND** (T is of size N)

- $A_i \rightarrow A_j$

- ($exchange\_not\_occur$ = FALSE) **AND** ($i < N$) **AND** $((T'(i) > T'(i+1)) \rightarrow$ ($T'(i)$ is exchanged with $T'(i+1)$)
- ($exchange\_not\_occur$ = FALSE) **AND** ($i \geq N$) $\rightarrow$ ($T'(i)$ is stored)

□ Other Proof Technique:

The **logical proof** ignores the structure of the programming language employed.

$\Downarrow$

The *logical proof* proves that the components design is correct, not necessarily its implementation.

$\Downarrow$

symbolic execution technique

$\Downarrow$

is a simulated execution of the code using symbols instead of data variables.

$\Downarrow$

The program is viewed as having an input state dermined by the input data and conditions

$\Downarrow$

As each line of the code is executed, the technique checks to see whether its state has changed

$\Downarrow$

Each state change is saved.

$\Downarrow$

Program execution is viewed as a series of state changes.

$\Downarrow$

The final state of each path should be an output state.

$\Downarrow$

The program is correct if each possible input state generates the proper ouptut state.

$\Downarrow$ e.g.

- $a = b + c$;
- **if** $(a > b)$ $\quad$ $task\_1()$;
- **else** $\quad$ $task\_2()$;

$\Downarrow$

The condition $a > d$ is treated as two possible states:

$a > d$ is *false* $\quad$ and $\quad$ $a > d$ is *true*

instead of testing a large number of possible values

□ Automated Theorem Proving:

- The ideal theorem prover would read in any program and
- produce as its output

  either
  - ⋆ a statement confirming the code's correctress

  or
  - ⋆ the location of a fault

$\Downarrow$ but unfortunately

- it is equivalent to the *halting problem for Turing machine* (i.e., no solution)

□ Proof or Test?

- proof
  - how a program will work in a hypothetical environment described by the design and requirment.
- testing
  - how a program works in its actual environment

□ Choosing test cases:

- A test case:
  - a particular choice of input data to be used in testing the program.
- A test:
  - a finite collection of test cases.

$\Downarrow$ But,

how do we choose test cases to know the program works correctly, not only for the test cases but for all input?

$\Downarrow$

Separate the input into equivalence classes

$\Downarrow$ so that,

$\Downarrow$

the classes should meet the criteria:

- Every possible input belongs to one of the classes.
  - ⋆ The classes should cover the entire set of input data.
- No input datum belongs to more than one class.
  - ⋆ The classes should be disjoint.
- If the execution code demonstrates a fault when a particular class member is used as input,

  $\Downarrow$ then

  the same fault can be detected using any other member of the class as input.
  - ⋆ any element of the class represents all elements of that class.

    (or at least)
  - ⋆ the probability should be high that every other element in that class will reveal the same fault.

□ Choosing test cases: cont'd

Examples:

- Input of three real numbers a, b, and c.
  - $\star\ a > b > c, \quad b > c > a, \quad c > b > a, \cdots$
  - $\star\ a$ >0, b >0, c >0, $\quad a$ <0, b >0, c >0, $\quad \cdots$
  - $\star$ What if they are coeficients of a quadratic equation?
- For a positive input value:
  - · a very large positive integer
  - · a positive integer
  - · a positive, fixed-point decimal
  - · anumber greater than 0, but less than 1
  - · zero
  - · negative number

  (Some data are purposely chosen to be improper: we test them to check that the code handles incorrect data gracefully.)

□ Choosing test cases: cont'd

- Furthermore, we can add test cases
  - ⋆ to test all branches, or
  - ⋆ to exercise as many paths as possible.

- If loops are involved, we may include test cases that
  - ⋆ loop once;
  - ⋆ many times;
  - ⋆ not at all.

- If the algorithm includes the trigonometric function,
  - ⋆ test extremities such as 0, 90, 180, 270, and 360 degrees.

□ Test Thoroughness

To convince that the test exhibit all possible behavior:

- Statement testing
  - Every statement in the component is executed at least once in some test.
- Branch testing
  - For every decision point in the code, each branch is chosen at least once in some test.
- Path testing
  - Every distinct path through the code is executed at least once in some test.
- Definition-use path testing
  - Every path from every definition of every variable to every use of that definition is exercised in some test.
- All-uses testing
  - The test set includes at least one path from every definition to every use that can be reached by that definition.

□ How about random test?

- Ntafos's comparison (1984)

  on seven mathematical programs with knwon faults.

Random	Branch	All-uses
79.5%	85.5%	90.0%

□ Integration Testing:

After being satisfied that individual components are working correctly

$\Downarrow$

we must combine them into a working system.

$\Downarrow$ But how?

- Big-bang Integration
  - After all components are tested in isolation, they are mixed together as the final system and tested if it works.
  - Many programmers use this for small systems, but it is not proctical for large ones.
    * it is difficult to find the cause of any failure.
    * interface faults cannot be distinguished easily from other types of faults.
- Bottom-up Integration
  - The lowest-level components should be tested by **component driver**, a special code to call a particular component and passes a test case to it.
  - Though the top level components are usually the most important but the last to be tested.
  - It is difficult when much of the system's processing depends on timing.

- Top-down Integration
  - using **stub**, a special purpose program to simulated the activity of the missing component, because a component being tested may call that is not yet tested;
  - a very large number of stubs may be required when the lowest system level contains many general-purpose routines.
- Sandwich Integration
  - is made up of three layers:
    * Top layer $\Rightarrow$ topdown integration
    * Target layer
    * Bottom layer $\Rightarrow$ bottom-up integration

□ Testing Tools:

- Code Analysis Tools
  - Static vs. Dynamic Analysis
    - static ⇒ analysis before it is run.
    - dynamic ⇒ during the program is running.
  - Static Analysis
    - Code analyzer
      - check syntax or items not been defined.
      - can generate a symbol table to record where a variable is first defined and when it is used.
    - Structure checker
      - generate a graph depicting the logic flow.
      - can read a program and (1) check all loops marking statements that are never executed and (2) note the presence of branches from the middle of a loop.
      - list all possible path through the program
    - Data analyzer
      - reviews (1) data structures and (2) data declarations, and
      - notes conflictions of data definitions and illegal usage.
      - notify us when a denominator may be set to zero
      - check whether arguments to a subroutine are passed properly.

* Link cheker
  - review component interfaces and note improper linkage among components.
* Sequence checker
  - checks whether coded in the proper sequence of events.
  - The input/output components may be checked here to see if the events are coded in the proper sequence,

    $\Downarrow$ e.g.

    all files are opened before they are modified?
* what else by static analysis?
  - information about fan-in and fan-out;
  - the number of operators and operands';
  - the number of decision points;
  - several measures of the code's structural complexity, and
  - warn us about possible faults to be fixed before formal tests are run.

- Dynamic Analysis
  * during several parallel operations are performed concurrently:
  * captures the states during a program run by a series of *snapshots*;
  * lists the number of times
    · a component is called
    · a line of code is executed
  * reports whether a decision point branched in all directions;
  * gives statistics such as the percentage of statements, paths and branches that have been covered by the set of test cases run.
  * Statistics can be generated as for particular variables: their first value, last value, minimum, maximum, $\cdots$.
  * Breakpoints can be defined so that when a variable attains or exceeds a certain value, the occurence is reported, or stopped allowing the tester to check the conditions.
  * Control flow can be traced backward or forward from a break point and can be tested with different data.

- Test Execution Tools
  - Capture and replay
    * capture key strokes, input, and responses;

      $\Downarrow$
    * compare the responces with expected one for the test-cases specified;
    * analyse the discrepancy going back to its cause

      $\Downarrow$ After fixing a fault,
    * check if the fault has been corrected without introducing new fault
  - Automatic generation of drivers
    * such that it enables us to
      1. set all appropriate state variables to prepare for a given test case, and then run the test case;
      2. simulate keyboard input and other data-related response;
      3. compare actual outcome to expected outcome and report diferences;
      4. track which paths have been traversed during the run;
      5. set variables to prepare for the next test case;
      6. interact with a debugger, so that faults can be traced and fixed

- Automated testing environment
  - These tools above can be connected to
    - a testing database;
    - measurement tools,
    - code analysis tools,
    - text editors
    - simulation and modeling tools

    so that it automates the test as much as possible.

- Test Case Generators
  - Structural test case generators
    - is based on
      - the structrur of the source code.
    - lists test cases for path, branch, or statement.
  - Others
    - are based on
      - data flow;
      - functional testing (i.e., on exercising all possible states that affect the completion of a given function);
      - the state of each variable in the input domain.
  - Random sets of test data genarators

Note: Finding evidence of a fault is not the same as locating the fault itself. Test will always involve the manual effort required to trace a problem back to itys root cause. The automation assists but does not replace this human roll.

□ Testing Tools: summary

- Code Analysis Tools
  - Static Analysis
    * Code analyzer
    * Structure checker
    * Data analyzer
    * Link cheker
    * Sequence checker
  - Dynamic Analysis
- Test Execution Tools
  - Capture and replay
  - Automatic generation of drivers
  - Automated testing environment
- Test Case Generators

□ When to Stop Testing:

Establishing test objectives

$\Downarrow$

Designing test cases

$\Downarrow$

Writing test cases

$\Downarrow$

Testing test cases

$\Downarrow$

Executing tests

$\Downarrow$

Evaluating test results

$\Downarrow$But

when a test can be said to be over,

answering the software is free of faults and

performs the functions as specified in the requirements?

□ When to Stop Testing: cont'd

- Myres (1979) reports that:

  As the number of detected faults increases, the probability of the existence of more undetected faults increases.

- Fault Seeding (Mills, 1972)
  - One member of the test team intentionally *seed* a known number of faults in a program.

$$\Downarrow$$

  - The other members locate as many faults as possible.

$$\Downarrow$$

$$\frac{\text{detected-seeded-faults}}{\text{total-seeded-faults}} = \frac{\text{detected-nonseeded-faults}}{\text{total-nonseeded-faults}}$$

$$\Downarrow$$

  - But are seeded faults representative of the actual ones? (We don't know what are the typical faults before we've found them.)

$$\Downarrow$$

- Using two testing teams.
  - Use two independent test teams for the same program.

    * $x \Rightarrow$ the number of faults detected by G-1;
    * $y \Rightarrow$ by G-2;
    * $q \Rightarrow$ both by G-1 & G-2;
    * $n \Rightarrow$ total number of faults.

  - Effectiveness of test:

$$E_1 = x/n \qquad E_2 = y/n.$$

$$\Downarrow$$

  - But n is unknown to us.

$$\Downarrow \text{ so,}$$

  - G-1 found q of the y faults that G-2 found, i.e.

$$E_1 = q/y$$

$$\Downarrow$$

$$n = q/E_1 \cdot E_2$$

$$\Downarrow \text{ e.g.,}$$

$$x = 25, \quad y = 30, \quad q = 15.$$

- Using Confidence.

  - the likelihood that the software is fault free.

$$\Downarrow$$

  - a program is fault free with a 95% level of confidence. (the probability that the software has no faults is 0.95)

    - $S \Rightarrow$ the number of faults seeded;
    - $s \Rightarrow$ the number of seeded faults detected;
    - $N \Rightarrow$ the number of actual faults to be claimed (we claime that the code has only N faults)

$\Downarrow$ The confidence level is given as:

$$c = \frac{\binom{S}{s-1}}{\binom{S+N+1}{N+s}}$$

assuming that all faults have an equal probability of being detected, not likely to be true though.

$$\Downarrow$$

– Test the program until we have found all S of the seeded faults.

$$\Downarrow$$

– The confidence level is given as:

$$c = s/s + 1$$

$$\Downarrow$$

If required level of confidence is 98%

$$c = s/s + 1 = 98/100 \quad \Rightarrow \quad s = 49$$

Chapter 7

Testing the System

Requirment Analysis
$\Downarrow$
System Design
$\Downarrow$
Program Design
$\Downarrow$
Unit/Integration Testing
$\Downarrow$
System Testing
$\Downarrow$
Maintenance

□ Soureces of Faults:

- Requirement Analysis
  - Incorrect, missing or unclear requirements/translation;
- System Design
  - Incorrect/unclear design specification
- Program design
  - Incorrect/unclear design specification
  - Misinterpretation of system design
- Program Implementation
  - Misinterpretation of program design
  - Incorrect documentation
  - incorrect syntax/semantics
- Unit/Integration Testing
  - Incomplete test procedures
  - New faults introduced when old ones corrected
- System Testing
  - Incomplete test procedures
- Maintenance
  - Incorrect user documentation
  - Human mistakes
  - New faults introduced when old one corrected
  - Changes in requirements

□ System Test: Function → Performance → Acceptance → Implementation

- **Function Test**
  - checks if the integrated system performs its functions as specified in the requirements.

    $\Downarrow$ called

  (functioning system)

- **Performance Test**
  - compares the integrated components with the nonfunctional system requirements, which includes
    * security, accuracy, speed, and reliability
  - At this point, the system works as the designer's interpretation of the requirements specification.

    $\Downarrow$ called

  (verified system)

  - further compares the system with the customer's expectations by reviewing the requirements definition document

    $\Downarrow$ if we are satisfied it is called

  (validated system)

- **Acceptance Test**
  - assures the customers that the system they requested is the system that was built.

    $\Downarrow$ if we are satisfied it is called

  (accepted system)

- **Installation Test**
  - is run in its actual environment, and
  - allows users to exercise system functions.

□ For example:

- A **function test** of a bank account package
  - verifies if the package can correctly
    - credit a deposit;
    - enters a withdrawal;
    - calculates interst;
    - prints the balance;
    - etc.
- A **performance test**
  - evaluates
    - the speed with which calculations are made;
    - the precision of the computation;
    - the security precautions required;
    - the response time to user inquiry.
- What about in **acceptance test**?
- What about in **installation test**?

□ Incremental Test:

A system can be regarded as a nested set of subsystems.

$\Downarrow$

Each level is responsible for performing at least the functions of those subsystems.

$\Downarrow$ Similarly,

We can divide the test into a nested sequence of subsystems,

$\Downarrow$ and

perform the test on one subsystem at a time.

$\Downarrow$

Subsystem division is based on functionality,

$\Downarrow$ e.g.,

- the most critical functionality
- desirable ones
- least needed ones

$\Downarrow$

$\Downarrow$

Each larger subsystem contains all the susystems preceding it.

$\Downarrow$ For example,

the star network whose center receives messages from several terminals, and each of which captures data from sensors and transmits them to the central one for processing.

$\Downarrow$ Devide into 6 levels:

1. test the central computer's general functions;
2. test the central computer's message translation function;
3. test the central computer's message assimilation function;
4. test each terminal in the stand-alone mode;
5. test each terminal's message sending function;
6. test central computer's message receiving function

□ **Function Testing** (more in detail)

- We now need not know which component is responsible,

  $\Downarrow$ rather

  we must know what the system is to do.

- test one function at a time

- functions may be defined in a nested way, such as:
  - · function 1
    - · function 1.1
      - · function 1.1.1
      - · function 1.1.2
      - · function 1.1.3
    - · function 1.2
  - · function 2
  - · . . .

- Test cases should be from the requirements document

$\Downarrow$ A tool (Elmendorf 1973, 1974)

converts the natural language of requirements definitions to a formal specification that can be used to enumerate test cases.

$\Downarrow$

by examinning the semantics of the requirements

$\Downarrow$

it restates them as logical relationships between inputs (**cause**) and outputs/transformation (**effects**)

$\Downarrow$

A Boolean graph

□ To Create a Cause and effect graph

1. The requirements are separated so that each requirement describes a single function;
2. All causes and effects should be described;
3. The numbered causes and effects become nodes of the graph;
4. Place causes on the left-hand side of an arc and effects on the right.

$\Downarrow$ using relations:

- if A then B;
- if A and B then C;
- if A or B then C;
- if not (A and B) then C;
- if not (A or B) then C;
- if neithe (A or B) then C;
- if not A then B;
- exactly one of A and B can be invoked;
- at most one of A and B can be invoked;
- at least one of A and B can be invoked
- Effect A masks the observance of effect B
- The invocation of A requires the invocation of B

$\Downarrow$

- Decision table
  - raw $\Rightarrow$ each cause/effect
  - column $\Rightarrow$ testcases
  - entity for cause $\Rightarrow$ T, F or # (True, False, Dont-care)
    entity for effect $\Rightarrow$ A or P (Absence, Presence)

□ Performance Tests:

Types of Performance Tests

- Stress tests
  - test the upper bound limit case of the system.
  - For example, see if
    * if the requirements state that a system is to handle up to specified number of devices or users, a stress test evaluates system performance when all those devices or users are active simultaneously.

- Volume tests
  - address the handling of large amounts of data in the system.
  - For example, see if
    * data structures such as queues or stacks have been defined to be large enough to handle all possible situations.
    * fields, records and files can accommodete all expected data, if it's a data base system.
    * the system reacts correctly when data sets reach their maximum size.

- Configuration tests
  - analyze the various software and hardware configurations specified in the requirements.
- Compatibility tests
  - are needed when a system interfaces with other systems
  - e.g.,
    * if the system is to communicate with a large database, the speed and accuracy of data retrieval should be tested.
- Regression tests
  - are needed when the system is replacing an existing one.
    * it should guarantee that the new system's performance is at least as good as toe old one.
- Security tests
- Timing tests
  - evaluate time to respond to a user
  - if a transaction must take place within a specified time, the test performs that transaction and verify that it fulfills the requirement.
  - the test should be perfromed with stress test so that it garantees the performance even when the system is extremely active.

- Environmental tests
  - look at the system's ability to perform at the installation site.
  - including tolerances for heat, humidity, motion, chemical presence, electrical or magnetic fields, disruption of power and so on;
- Quality tests
  - evaluate the system's reliability, maintainabiliyt and availability.
  - including mean time to failure and mean time to repair, or average time to find and fix a fault
- Recovery tests
  - adress response to the presence of faults or to the loss of data, power, devices or sercices.
  - We also subject the system to a loss of system resources and see if it recovers properly.
- Maintenatnce tests
  - address the need for diagnostic tools and procedures to help in finding the source of problems.
  - supplying diagnostic programs, memory maps, traces of transaction, circuit diagrams, and so on.
  - it should be verified that the aids exist and they function properly.

- Documantation tests
- Human factor tests
  - investigate the user interface.
  - examine display screens, messages, report formats, etc.
  - also called usability tests

□ Reliability, Availability, and Maintainability:

- definiton:
  - software reliability
    - the probability that a system will operate without failure under given conditons for a given time interval
  - software availability
    - the probability that a system is operating successfully according to specification at a given point in time,
  - software maintainability
    - the probability that, for a given condition of use, a maintenace activity can be carried out within a stated time interval and using stated procedures and resources

- Can we predict the length of time to the next failure?
  - Prot interfailure times to see if reliability is growing.
    - type-1 uncertainty
      - unpredictable uncertainty
      - We cannot predict which fault will trigger the next failure.
    - type-2 uncertainty
      - reflects our lack of knowledge

- How to measure them?
  - assign value from 0 (unreliable, unavailable, or unmaintainabel) to 1 (completely reliable, alwqays available, and completely maintainabel)
  - mean-time-to-failure (MTTF) of interfailure time $t_1, t_2, \cdots, t_{i-}$
  - mean-time-to-repair (MTTR)
  - mean-time-between-failure (MTBF)
    - MTBF = MTTF + MTTR
  - Shooman (1993)
    - R = MTBF / (1 + MTBF)
    - A = MTBT / (MTBF + MTTR)
    - M = 1 / (1 + MTTR)

- Reliability Stable/Growing
  - if the interfailure times stay the same ⇒ stable

    if the interfailure times increases ⇒ growing
  - if we can define a probability density function $f(t)$, the likelifood that a software will fail in a time interval t_1, t_2 is $f(t)dt$

□ Reliability Prediction Model:

- The simplest model:
  - $t_i = (t_{i-1} + t_{i-2})/2$
  - $t_i = (t_{i-1} + t_{i-2} + \cdots + t_{i-n})/n$

- The Jelinski-Moranda Model
  - under two assumption of
    - ∗ There is no type-2 uncertainty
      (fault corrections are perfect: fix the fault causing the failure without introducing no new faults)
    - ∗ fixing any fault contributes equally to improving the reliability

- The Musa Model

- The Littlewood Model

□ System Test:

- **Function Test**
  - checks if the integrated system performs its functions as specified in the requirements.
- **Performance Test**
  - compares the integrated components with the nonfunctional system requirements, which includes
- **Acceptance Test**
  - assures the customers that the system they requested is the system that was built.
- **Installation Test**
  - is run in its actual environment, and

□ Acceptance Testing:

- Customer's chance to verify that what was wanted is what was built.

- Test are written, conducted and evaluated by the customers (with assistance from the developers only when the customer requests an answer to a technical question.)

- Three ways the customer evaluate
  - A benchmark test
  - A pilot test
  - A parallel test

□ Three ways the customer evaluate:

- **A benchmark test**
  - When a customer has special requirments.
  - Two or more development teams are asked to produce systems.
  - One system will be chosen for purchase, based on the success of benchmark test.

    $\Downarrow$

    * how the systems meet the benchmark criteria under a set of test cases that are prepared by the customer.
- **A pilot test**
- **A parallel test**

- **A pilot test**
  - installs the system on an experimental basis;
  - relying on the everday working of the system to test each function;
  - much less fromal and structured than a benchmark test;
  - sometimes we pilot the system before the customer runs the real pilot test
    * our in-house test $\Rightarrow$ Alpha test
    * the customer's pilot $\Rightarrow$ Beta test
  - this approach is common when systems are for a wide variety of customers;
  - we must choose beta test sites so that they represent all kinds of system usage
- **A parallel test**
  - if a new system is replacing an existing one
  - new one should be operated in parallel with the old one.

□ Installation Testing:

- The final round of testing.
- involves installing the system at user sites.
- We attach the proper number and kind of devices to the main processor and establish communications with other systems.
- Tests focus on
  - Completeness of the installed system;
  - Verification of any functional or non-functional characteristics that may be affected by site conditions.

- When building large systems, there is sometimes a long lag between the initial specification and the first viewing of even part of a system

  $\Downarrow$

  Customer's needs may change

  $\Downarrow$

  Aftter acceptance test, which requirements are not satistified and which must be deleted revised or added because of changing needs.

  $\Downarrow$

  When the customer is satisfied with the results

  $\Downarrow$

  testing is completed

  $\Downarrow$

  The system is formally delivered

□ Test Documentation:

- **Test Plan**
  - describes the system itself and the plan for testing all functions and characteristics.
- **Test Specification and evaluation**
  - details each test and,
  - defines the criteria for evaluating each feature.
- **Test Description**
  - presents the test data and procedures for individual tests.
- **Test Analysis Report**
  - describes the results of each test
- **Problem Report**
  - . . .

□ An Example of a Test Description:

INPUT DATA:

Input data are to be provided by the LIST program. The program generates randomly a list of N Words of alphanumeric characters; each word is of lenght M. The program is invoked by calling

RUN LIST(N, M)

in your test driver. The output is placed in global data area LISTBUF. The test datasets to be used for this test are as follows:

CASE 1: Use LIST with $N = 5$, $M = 5$.

...

CASE 6: Use LIST with $N = 150$, $M = 10$.

INPUT COMMANDS:

The SORT routine is invoked by using the command:

RUN SORT(INBUF,OUTBUF) or
RUN SORT(INBUF)

OUTPUT DATA

If two parameters are used, the sorted list is placed in OUTBUF. Otherwise, it is placed in INBUF.

SYSTEM MESSAGES:

During the sorting process, the following message is displayed:

"Sorting... Please wait..."

Upon completion, SORT displays the following message on the screen:

"Sorting completed"

To halt or terminate the test before the completion message is displayed, press CONTROL-C on the keyboard.

□ An Example of a Test Script:

- Press function key 4: Access data file.
- Screen will ask for the name of the date file. Type 'sys:test.txt'
- Menu will apear, reading
  - · delete file
  - · modify file
  - · rename file

  Place cursor next to 'modify file' and press RETURN key.
- Scree will ask for record number. Type '4017'.
- Screen will fill with data field for record 4017:

  Record number: 4017

  X: 0042 Y: 0036

  Soil type: clay

  ...
- Press function key 9: modify
- Entries on screen will be highlighted. Move cursor to VEGETATION field. Type 'grass' over 'kudzu' and press RETURN key.
- Press function key 16: Return to previous screen.
- Menu will appear. reading
  - · delete file
  - · modify file
  - · rename file

  To verify that the modification has been recorded, place cursor next to 'modify file' and press RETURN key.
- Screen will ask for record number, Type '4017'.

...

□ Problem Report:

- Questions that are to be asked
  - Where did the problem occur? (locations)
  - When did it occur? (timing)
  - What was observed? (symptom)
  - What were the consequences? (result)
  - How did it occur? (mechanism)
  - Why did it occur? (cause)
  - How much was the user of business affected? (severity)
  - How much did it cost? (cost)

- Examples of answers from **developer's understanding**

- **Location:** module number;
- **Timing:** phases of development during which fault was created, detected, and corrected;
- **Symptom:** error message reported, or activity that revealed fault;
- **Result:** failure caused by the fault;
- **Mechanism:** how the source was created, detected and corrected;
- **Cause:** type of human error that led to the fault;
- **Severity:** refers to sevrity of a failure;
- **Cost:** time or effort to locate and correct (or, analysis of cost if fault had bee identified earlier)

- Examples of answers from **user's view**

- **Location:** installation where the failure was observed;
- **Timing:** clock time;
- **Symptom:** error message or indication of failure;
- **Result:** description of failure, such as
  - operationg system crash
  - service degraded
  - loss of data
  - wrong output
  - no output
- **Mechanism:** cjaom pf events, including keyboard commands and state data leading to failure;
- **Cause:** reference to possible faults leading to failure;
- **Severity:** impact on business;
- **Cost:** cost to fix plus cost of lost business;

Chapter 8

Maintaining the System

□ Lehman's System Categorizaiton:

Why are some systems more prone to change than others?

- **S-systems** (Specifiable-)
  - Systems that are possible to be defined by a specification formally and completely .
  - The solution is well-known.

    $\Downarrow$

    The developer's concern is
    * not on the correctness of the solution,
    * but on the correctness of the implementation of the solution
  - If the world changes a completly new problem must be specified.

- **P-systems** (Practical-)
- **E-system** (Embedded-)

- **P-systems** (Practical-)
  - Systems whose requirments specification are written from the abstract description of the problem.
  - are based on a practical abstraction of the problem rather than on a completely defiend specification
  - When it's difficult/impossible to describe a realworld problem,

    or
  - when theoretical solution to a problem exists but implimenting the solution is impractical or impossible,

    $\Downarrow$

    we must develop an approximate solution that is more practical to build and use.
  - When the solution is unsuitable the problem abstraction may be canged and the requirements may be modified to make the solution more realistic.
  - The solution is acceptable if the results make sense in the world where the problem is embedded,
    * while in a S-system the solution is acceptable if the specifications are correct.

- Many things can change in a P-system when the solution is compared with the world,
  * the abstraction may change
  * the requirements may need to be altered
  * the implementation may be affected.

- The system resulting from the changes is
  * not a new solution to a new problem
  * but a modification of the old solution to find a better fit to the problem.

- **E-system** (Embedded-)
  - The system that changes as the world changes, and

  - are likely to undergo almost constant change, while
    * S-systems are unlikely to change and
    * P-systems are subject to incremental change,

□ How much change can we expect?

- S-systems will have little of no change.
- P-systems will have much more.
- E-systems are likely to change continually.

$\Downarrow$ average

- the 80-20 rules
  (20% of the effort is in development and 80% is in maintenance.)

$\Downarrow$

- maintenance stage = evolutionary

□ Evolve or Discard?

- Is the cost of maintenance too high?
- Is the system reliability unacceptable?
- Can the system no longer adapt to further change within a reasonable amount of time?
- Is system performance still beyond prescribed constraints?
- Are system functions of limited usefulness?
- Can other systems do the same job better, faster, or cheaper?
- Is the cost of maintaining the hardware great enough to justify replacing it with cheaper, newer hardware?

$$\Downarrow$$

- cost from seystem creation to retirement = life-cycle cost

□ **Lows of Software Evolution** (Lehman, 1980)

- **Continuing change**
  - A program becomes progressively less usuful. The decay continues until it is more cost-effective to replace the system with a recreated version.
- **Increasing complexity**
  - As an evolving program is continually changed, its structure deteriorates and its complexity increases (unless work is done to maintain or reduce it).
- **Fundamental law**
  - Program evolution is subject to a dynamic that makes the programming process self-regulating with statistically-determinable trends and invariance.
- **The Law of Conservation I**
  - During the active life of a program, the global activity rate in a programming project is statistically invariant.
- **The Law of Conservation II**
  - During the active life of a program, the rate of changes, addition, and deletions of the successive releases of an evolving program is statistically invariant.

□ **Lows of Software Evolution** (Lehman, 1980) cont'd

- **The 1st law says that:**
  - large systems are never complete but they continue to evolve;
  - the systems grow as we add more features, apply more constraints, interact with other systems, support a larger number of users, and so on.
  - they also change because their environment changes;
  - the are ported to other platforms or rewritten in new languages.

- **The 2nd law is partly due to:**
  - our hasty patches to fix problems;
  - the lack of time to maintain the elegance of a design or the consistency of approach.

- **The 3rd law says:**
  - software systems exhibit regular behaviors and trends that we can measure and predict.

- **The 4th law says:**
  - there are no wild or wide fluctuations in organizational attributes, such as productivity.
  - At some point, resoruces and output reach an optimum level, and adding more resources does not change the output significantly.

- **The 5th law says:**
  - After a while, the effects of subsequent releases make very little difference in overall system functionality.