CS504 QUIZ 1 SOLVED

1.Names representing methods and functions should be ___ and written in mixed case starting with ___ case.

a) Noun --- lower

b) Verb --- lower

c) Noun --- upper

d) Verb --- upper

2. In case of file servers, clients requests selected records from a ___ and the server transmits records to client over the network.

a) Local Memory

b) Network

c) Database

d) File

3. Which of the following is among ten things, which the basic template of GOF design pattern includes.

a) IQ

b) Context

c) Performance

d) Requirements

4. Which one of these represents Kruchten's 4+1 architectural view model?

a) Logical view, Process view, Physical view, Development view, Use case view

a) Logical view, Dynamic view, Physical view, Development view, Use case view

a) Logical view, Process view, Physical view, Sequence view, Use case view

a) Dynamic view, Process view, Physical view, Development view, Use case view

5. C++ header files should have the extension ___ source files can have the extension ___

a) .c and .h

b) .cc and .h

c) .h and .cpp

d) All of the given

6. Code should not be:

a) commented

b) indented

c) cryptic

d) aligned

7. Variables should be initialized where they are ___ and they should be declared in the ___ scope possible.

a) defined and smallest

b) declared and medium

c) defined and medium

d) declared and smallest

8. It ensures that the class only has one instance and provides a global point of access to it.

a) Singleton pattern

b) Observer pattern

c) Real Pattern

d) None of the given

9. Which of the following is used for multi-level commenting?

a) // Comment

b) /* Comment */

c) (Comment)

d) */ Comment /*

10. Which of the following shows a commented statement in C++

a) # Ans = first + second

b) // Ans = first + second

c) \\ Ans = first + second

d) /# Ans = first + second

11. Fat Client Model is one of the configurations of ___ Model.

a) Data-centered

b) Layered

c) Reference

d) Client Server

12. Data-Centered Architectural Style is also called ___

a) Repository Model

b) Client-Server Model

c) Sub System Model

d) Reference Model

13. When programmers use shortcuts and cryptic codes in their program, ___ becomes a major problem

a) Maintenance

b) Reusability

c) Availability

d) Readability

14. In Client-server model, the application is modeled as a set of ___ that are provided by servers.

a) Requests

b) Protocols

c) Services

d) Requirements

15. Vertical partitioning divides the architecture's application from a ___ making perspective.

a) Decision

b) Design

c) Conclusion

d) Move

16. Which one is correct?

a) double total = 0.5;

b) double total = .5;

c) double total = .50;

d) all of the given

17. Which one of these is a correct shortcut used by the programmer?

a) x*=a

b) x=*a

c) x*a=

d) =a*x

18. Pattern are devices that allow programs to share knowledge about their ___

a) Code

b) Design

c) Analysis

d) Implementation

19. Which indent size eliminates the chance of code lines splitting?

a) 2

b) 4

c) 3

d) 6

20. Which of the following is not a possible server in clients server environment?

a) Database Server

b) Transaction Server

c) File Server

d) Time Server

-----provides a unified interface to a set of interfaces in a sub-system. Select correct option:
Observer Pattern
Singleton Pattern
Façade Pattern
All of the above

"is" prefix should be used for------ variables and methods. Select correct option:
General
Boolean
Constant
None of the given

using proper paranthesis normally makes the code.
easy to read
easy to understand
less ambiguous
All of the given options

The use of comments should be minimized by making the code self-documenting by appropriate name choices and an explicit logical structure.
TrueFalse

Bit fields allow the packing of data in a structure. using Bit fields we can:
Read 9 bit integers
Avoid memory leakages
Avoid memory overflow
Avoid syntax errors

1) x = (a + 2 > 3)? a : a-1 ; 2) if((a + 2)> 3) x = a; else x = a - 1;
Statement (2) is more complex than (1) Statement (2) is more complex than (1)
Both statements are very complex
None of the given option

Façade Pattern provides a unified interface to a set of interfaces in a sub-system. Select correct option:
True
False
[Make the code easy to modify Does not effect understandablity

Complex expressions:
Select correct option:
Make the code easy to modify
Make the code difficult to modify
Make the code easy to understand Does not effect understandablity

The terms get/set must be used where an attribute is accessed Select correct option:
Indirectly
Directly

In order to make a code more portable, Instead of using vendor specific language extensions, use _____ as much as possible
STL
ANSI

ISO
CMMI

Question # 7 of 10
Exception handling provides:
Portability mechanism
Code Usability mechanism
Error Handling mechanism
All of the given options

Question # 8 of 10
Bit fields are a convenient way to express many difficult operations. However, bit fields suffer from one problem
Lack of usability
Lack of security
Lack of performance
Lack of portability

Question # 9 of 10
if (e.Title() == "CEO" || e.Salary() > 10000) in the above statement if e.Title()=="CEO") is TRUE then
Second part is not evaluated
Second part is always evaluated
Second part must also be TRUE
Second part is not evaluated but e.Salary() is called

Question # 10 of 10
A test case involves
Input/output specification plus a statement of the function under test
Steps to perform the function
Expected results that the software application produces
All of the given options

A self documenting code is a code that explains itself without the need of comments and extraneous documentation, like _____
Select correct option:
Flowcharts
UML diagrams Process-flow state diagrams All of the given choices

1
which of the following statements are same in output: 1) a = a >> 2 2) a = a / 4 3) a = a * 2
Select correct option:

(1) and (3) only
(2) and (3) only
(1) and (2) only
All procduce the same result

he hardest single part of building a software system is deciding precisely ---------- to uild.

- ***what***
- How
- When
- Why

nteraction Diagrams depict the ---------- Behavior of the system

- Static
- **Dynamic**
- Active
- Non of the above

Identifier names also play a significant role in enhancing the -------- of a program. Select correct option:
Writ ability
Readability
Reliability
All of the given choices

1) x = (a + 2 > 3)? a : a-1 ; 2) if((a + 2)> 3) x = a;
else x = a - 1; Select correct option:
Statement (1) is more complex than (2) Statement (2) is more complex than (1) Both statements are very complex None of the given options

using proper paranthesis normally makes the code
Select correct option: easy to read
easy to understand less ambigous
All of the given options

Comments are not syntax checked
Select correct option:
TRUE
FALSE

_____ was the first pure Object Oriented language in which observer pattern was used in implementing its Model View Controller pattern
Select correct option:
Smalltalk
PASCAL JAVA C++

Flexibility helps in making the code easier to understand. Select correct option:
True false

Name is the initial thing to be written in the GOF Design Pattern Format. Select correct option:
True

False

Comments should NOT be indented relative to their position in the code Select correct option:
TRUE
FALSE

UML (unified modeling language) analysis modeling focuses on the _____ .
- behavioral model and environment model.
- behavioral model and implementation model.
- user model and environmental model
- **user model and structural model** APRACTITIONER'SAPPROACH

The data flow diagram must be augmented by descriptive text in order to describe the functional requirements for a software product.
- **True**
- False

Which of the following shows a commented statement in C++ Select correct option:
Ans = first + second
// Ans = first + second
\\ Ans = first + second /# Ans = first + second

Most software continues to be custom built because
- **Software is easier to build without using someone else's components.**
- Off the shelf software components are not commonly available
- Component reuse is common in the software world
- Reusable components are too expensive to use

Which of the items listed below is not one of the software engineering layers?
- Tools
- Process
- **Manufacturing**
- Methods

if (!(block < activeBlock)) is equvivalent to:

Select correct option:
if ((block < activeBlock)) if ((block == activeBlock)) if ((block >= activeBlock)) None of the given

no idea about this question

-----provides a unified interface to a set of interfaces in a sub-system. Select correct option:
Observer Pattern
Singleton Pattern
Façade Pattern
All of the above

A self documenting code is a code that explains itself without the need of comments and extraneous documentation, like _____
Select correct option:
Flowcharts
UML diagrams Process-flow state diagrams All of the given choices

Façade Pattern provides a unified interface to a set of interfaces in a sub-system. Select correct option:
True
False

which of the following statements are same in output: 1) a = a >> 2 2) a = a / 4 3) a = a * 2
Select correct option:
(1) and (3) only
(2) and (3) only
(1) and (2) only
All procduce the same result

Unrelated variables should be declared in the same statement. Select correct option:
True
False

Which of the following shows a commented statement in C++ Select correct option:
Ans = first + second
// Ans = first + second
\\ Ans = first + second /# Ans = first + second

Name is the initial thing to be written in the GOF Design Pattern Format. Select correct option:
True
False

Name is the initial thing to be written in the GOF Design Pattern Format. Select correct option:
True
False

Class variables should never be declared public. Public variables violate which of the following:
Select correct option: Information hiding
Encapsulation

Information hiding and Encapsulation
None of the given

The nominal case should be put in the if-part and the exception in the else-part of an if statement. Select correct option:
TURE

FALSE

The three basic principles that guide maintainability are: (a)clarity (b)enhancement (c)flexibility (d)simplicity (
Select correct option: (a),(b) and (c) (a),(c) and (d) (b),(c) and (d)
All of above

DFD Notation contains
- Data Store
- Extenal Agents
- Processes
- **All of the given chap4 page 10**

A complex System evolves from a
- smaller system
- medium sysetem

- ► bigger system
- ► **non of the given chapter 6 p2 (simple system)**

Requirement engineering mainly deals with the ---------- of the system

- ► **definition phase chapter 3 p 2**
- ► development phase
- ► maintenance
- ► non of the above

Insufficient user involvement leads to ---------- products.

- ► **unacceptable chapter 3 p 5**
- ► ambigious
- ► dissatisfaction of customer to
- ► non of the above

Which one is not the purpose of Interaction Diagrams ?

- ► Model interactions between objects

The Use case diagram does not show which actors interact with each use case Select correct option:

TRUE
FALSE
A Process in Data Flow Diagram (DFD) represents
Select correct option:

Flow of data
Transformation of data
Storage of data
An external agent

A context diagram is used
Select correct option:

as the first step in developing a detailed DFD of a system in systems analysis of very complex systems
as an aid to system design
as an aid to programmers
In Data Flow Diagram (DFD), one data store can directly copy the data from another data store Select correct option:

TRUE
FALSE
3:53 PM
The system model template contains which of the following elements Select correct option:

input output output
all of the given
To construct a system model the engineer should consider one of the following restraining factors? Select correct option:

assumptions and constraints
budget and expenses
data objects and operations schedule and milestones
In data flow diagram (DFD), Create, Update, Delete and Read operations are normally called: Select correct option:
CRUD operations
DURC operations
RUDC operations
CDUR operation

The "extends" relationship is kind of a generalization specialization relationship. Select correct option:
TRUE
FALSE

Question # 1 of 10 (Start time: 04:00:12 PM) Total Marks: 1
The Use case diagram does not show which actors interact with each use case Select correct option:
TRUE
FALSE
There are some types of requirements that can not be documented in the use cases. Select correct option:
TRUE
FALSE

Question # 3 of 10 (Start time: 04:01:01 PM) Total Marks: 1
Arranging information in tabular form makes it easy to read, understand and comprehend as compared to streams of text.
Select correct option:
TRUE
FALSE

Quiz Start Time: 04:00 PM Time Left 88 sec(s)
Question # 4 of 10 (Start time: 04:01:25 PM) Total Marks: 1
To construct a system model the engineer should consider one of the following restraining factors? Select correct option:
assumptions and constraints
budget and expenses
data objects and operations schedule and milestones

Question # 7 of 10 (Start time: 04:03:01 PM) Total Marks: 1 The best way to conduct a requirements validation review is to Select correct option:
examine the system model for errors
have the customer look over the requirements
send them to the design team and see if they have any concerns
use a checklist of questions to examine each requirement

Question # 8 of 10 (Start time: 04:04:04 PM) Total Marks: 1
Data Flow diagram (DFD) does not capture control flow information, it just shows the flow of the data in a system.
Select correct option:
TRUE
FALSE

Question # 9 of 10 (Start time: 04:04:38 PM) Total Marks: 1
GUI stands for:
Select correct option:
Genaric User Interface
Graphical User Interface
Genaric User Interaction
Graphical User Interaction

Question # 10 of 10 (Start time: 04:05:00 PM) Total Marks: 1
System models are built to allow the system engineer to evaluate the system components in relationship to one another
Select correct option: True
False

DFD Notation contains

- Data Store
- Extenal Agents
- Processes
- **All of the given**

A complex System evolves from a
- ► smaller system
- ► medium sysetem
- ► bigger system

- ► **non of the given**

Question No: 8 (Marks: 1) - Please choose one
Requirement engineering mainly deals with the ---------- of the system
- ► **definition phase**
- ► development phase
- ► maintenance
- ► non of the above

Insufficient user involvement leads to ---------- products.
- ► **unacceptable**
- ► ambigious
- ► dissatisfaction of customer to
- ► non of the above

Which one is not the purpose of Interaction Diagrams ?
- ► Model interactions between objects

- ► Assist in understanding how a system (a use case) actually works
- ► Identify responsibilities/operations and assign them to classes
- ► **Identify dependencies among objecs**

Which one is not a type of messages which Sequence Diagrams Depict
- ► Synchronous
- ► Asynchronous
- ► Create
- ► **Update**

The two main documents produced during this phase are Requirement Statement and Requirement Specification. They are also called Requirement Definition and ----------
-
- ► **Functional Specification**
- ► Mathematical specification
- ► System Specification
- ► None of the given

Project ---------- defines the concept and range of the proposed solution, and limitations identify certain capabilities that the product will not include
- ► **Scope**
- ► Agreement
- ► Plan
- ► None of the given

A change becomes ---------- because of close presence of data and functions
- ► **Localized**
- ► Private

► Global
► Accessible

A complex System evolves from a
► **smaller system**
► medium sysetem
► bigger system
► non of the given

Coupling is a measure of ----------of a module or component.
► **Independent**
► Dependent
► Closeness
► All of the given

Software Maintenance phase involves
► **Debugging**
► Adding new features
► making changes
► all of the given

Interaction Diagrams depict the ---------- Behavior of the system p23 chapter7
► Static
► **Dynamic**
► Active
► Non of the above

In Sequence Diagrams the time required by the receiver object to process the message is denoted by an----------
► **Activation Box**
► Simple Box
► Arrow
► Non of the above

A poorly designed interface can cause a user to make catastrophic errors is one of the motivations for GUI
► **True**
► False

Identifying System Features include ------
► log important information
► conduct business
► analyze business results
► **all of the above**

Establishing responsibilities for objects includes
► Generalization Relationships
► Specialization Relationships
► **all of the above**
► identifying Association relationships

Please choose one Which of the following is a fact finding method?
► Site visits
► **Prototyping**

► Study of similar systems
► All of given

The Object Oriented approach
► **improves the reusability of codes**.
► makes objects less independent.
► increases testing time.
complicates mapping of a system model to an implementation model

Client Server model tries to data and processing Distribute

Merge Clone Proceed

Anti- Patterns is another concept that corresponds to common in analysis and design.

Mistake Issues
Problems
All of the given

Three tier architecture contains layers
Presentation
Application
Database
All of the above

MVC stands for
Model View Controller
Modern View Center
Model View Center
Modern View Controller

Fat client model is one of the configurations of model

Data-Centered
Layered
Reference
Client Server

Description of communicating objects and classes that are customized to solved a general problem in a particular context is called
Design Pattern
System Patter System Design None of
the Given

In the N-tire Architecture, the idea is to enhance scalability and by distributing both data and the application using multiple server machines.
Usability
Performance

Interpretability
None of the given

Thin Client model is one of the configurations of ….. model

Data Centered
Client Server
Layered
Reference

Dynamic process model shows the process …….. of the system

Components
Objects
Structure
Linkage

It ensures that a class only has one instance and provides a global point of access to it.
Singleton Pattern
Observer Pattern Real Pattern None of the given

STL Stnads for ----------
Standard Template Library
Standard Type Link
Standard Tempo Line
None of the given

Three tier architecture contains ---------- layers
Select correct option:
Presentation
Application
Database
All of the above

Which of following is/are among ten things,which the basic template of GOF design pattern includes.
Select correct option:

Problem
Context
Forces
All of the given

Vertical partitioning is also known as…..

Select correct option:

Balancing
Mutating
Parallelizing
Factoring

Patterns are devices that allow programs to share knowledge about their ---------- ----.

Select correct option:

Code
Design
Analysis
None of the given

Client server model tries to …….data and processing.

Select correct option:
Distribute
Merge
Clone
Proceed

N-tier architecture stems from the struggle to find a ---------- between the fat-client architecture and the thin-client architecture.

Select correct option:

Concurrency
Distribution point
Middle ground
Similarity

Vertical partitioning divides the architecture application from a …… making perspective.
Decision
Design Conclusion Move

Distributing the responsibilities to different subsystems so that we get a software system which is easy to maintain, is called ……the architecture.
Select correct option:
Subtracting
Partitioning
Cloning
Balancing

The nominal case should be put in the if-part and the exception in the else-part of an if statement.

Select correct option:
TURE
FALSE

Charles Simonyi first discussed Hungarian Notation. He was of ------ .

Select correct option:
Microsoft
IBM
Dell
Cisco

The terms get/set must be used where an attribute is accessed Select correct option:
Indirectly
Directly

A self documented program/code contains the following attribute(s):
Size of each function Choice of variable
Choice of variable
All of the given choices

"is" prefix should be used for------ variables and methods.

Select correct option:
General
Boolean
Constant
None of the given

A function should not be larger in any case and should not exceed _____ in length . Select correct option:
Half page
One page
Two pages Three Pages

It ensures that a class only has one instance and provides a global point of access to it. Select correct option:
Singleton Pattern
Observer Pattern Real Pattern None of the given

A self documented program/code contains the following attribute(s): Select correct option:
Size of each function Choice of variable Choice of variable

All of the given choices

The use of comments should be minimized by making the code self-documenting by appropriate name choices and an explicit logical structure.

Select correct option:
TRUE
FALSE

Variables should be initialized where they are ------and they should be declared in the ------scope possible.

Select correct option: defined and smallest
declared and medium defined and medium
declared and smallest

Unrelated variables should be declared in the same statement.

Select correct option:
True
False
which of the following statements are same in output: 1) a = a >> 2 2) a = a / 4 3) a = a * 2

Select correct option:
(1) and (3) only
(2) and (3) only
(1) and (2) only
All procduce the same result

In _____The objects are organized in a horizontal line and the events in a vertical time line.
Sequence diagram

An object or class may further be classified on the basis of
- ► Behavior Driven attributes
- ► Data Driven attributes
- ► Responsibility Driven attributes
- ► **All of the above c**

A public Interface provides a way for with other Classes.
- ► Comminication
- ► Accessibility
- ► Reaching
- ► All of the above

By following modern system engineering practices simulation of reactive systems is no longer necessary

- ► True
- ► False

The state transition diagram

- ► depicts relationships between data objects
- ► depicts functions that transform the data flow
- ► indicates how data are transformed by the system
- ► indicates system reactions to external events

Control flow diagrams are

► needed to model event driven systems.

► required for all systems
► used in place of data flow diagrams
► useful for modeling user interfaces

A change becomes ---------- because of close presence of data and functions

► **Localized**
► Private
► Global
► Accessible

A complex System evolves from a

► **smaller system**
► medium sysetem
► bigger system
► non of the given

Coupling is a measure of ----------of a module or component.

► **Independent**

► Dependent

► Closeness

► All of the given

Software Maintenance phase involves

► Debugging

► Adding new features

► making changes
► all of the given

The hardest single part of building a software system is deciding precisely ---------- to build.

- **what**
- How
- When
- Why

Interaction Diagrams depict the ---------- Behavior of the system

- Static
- **Dynamic**
- Active
- Non of the above

In Sequence Diagrams the time required by the receiver object to process the message is denoted by an---

-----

- **Activation Box**
- Simple Box
- Arrow
- Non of the above

Names representing methods and functions should be----and written in mixed case starting with -----case.

Select correct option:

Noun—lower

Verb----lower

Noun ---upper

Noun----upper

Façade Pattern provides a unified interface to a set of interfaces in a sub-system. Select correct option:

True

False

A poorly designed interface can cause a user to make catastrophic errors is one of the motivations for GUI.

- **True**
- False

Most software continues to be custom built because

- Software is easier to build without using someone else's components.

It is not possible to use ordinary data flow diagrams to model the functional requirements of real-time systems.

- ► True
- ► **False**

- ► **Off the shelf software components are not commonly available**
  - ► Component reuse is common in the software world
  - ► Reusable components are too expensive to use

Which of the items listed below is not one of the software engineering layers?

- ► Tools
- ► Process
- ► **Manufacturing**
- ► Methods

An state transition can only occur when triggered by a(n)_____

- ► actor
- ► collaboration attempt
- ► event
- ► none of the given

The evolution of the system depends on a sequence of state transitions triggered by a set of coming events.

A change becomes ---------- because of close presence of data and functions

- ► **Localized**
- ► Private
- ► Global
- ► Accessible

Once these requirements have been finalized, the ---------- process starts.

- ► **Construction**
- ► Development
- ► Maintenance phase
- ► Construction and Development

Identifying Whole-Part structures (Aggregations) means what are my ----------

- ► **Components**
- ► Structures
- ► Objecs
- ► Classes

Floating point constants should always be written with decimal point and at least Select correct option:

one decimal two decimal three decimal

none of the given

MVC stands for ----------
Select correct option:
Model View Controller
Modern View Center
Model View Center Modern View Controller

In _____The objects are organized in a horizontal line and the events in a vertical time line.
Sequence diagram

Question No: 4 (Marks: 1) - Please choose one
An object or class may further be classified on the basis of
- ► Behavior Driven attributes
- ► Data Driven attributes
- ► Responsibility Driven attributes
- ► **All of the above**

A public Interface provides a way for with other Classes.
- ► Comminication
- ► Accessibility
- ► Reaching
- ► All of the above

A complex System evolves from a
- ► **smaller system**
- ► medium sysetem
- ► bigger system
- ► non of the given

Coupling is a measure of ----------of a module or component.

- ► **Independent**
- ► Dependent
- ► Closeness
- ► All of the given

Software Maintenance phase involves

- ► **Debugging**
- ► Adding new features
- ► making changes
- ► all of the given

Which of these people would not be likely to part of the FAST team?

- hardware and software engineers
- manufacturing representative
- marketing representatives
- **senior financial officers**

An state transition can only occur when triggered by a(n)_____

- actor
- collaboration attempt
- **event**
- none of the given

Identifying System Features include ------

- log important information
- conduct business
- analyze business results
- **all of the above**





https://vulmshelp.com

AL-JUNAID INSTITUTE GROUP

https://vulmshelp.com/

LMS handling service are available
PAID
Assignment, QUIZ, GDBZ and GRAND QUIZ SERVICE are available
0304-1659294

VULMSHELP.COM
JUNAID MALIK:0304-1659294

AL-JUNAID INSTITUTE GROUP

CS504 GRAND QUIZ

Modules with high cohesion and low coupling can be treated and analysed as

- White boxes
- **black boxes**
- grey boxes
- none of these

While establishing the services for an object, the following fundamental questions should be asked:

- Why does the system need this object anyway?
- What useful questions can it answer?
- What useful action can it perform?
- All of the given options.

----------is a role that each actor plays in the system under consideration.

- An act
- **A participant**
- A function
- None of the given

Any Engineering approach must be founded on organizational commitment to ----------.

- Cost
- Scheduling
- **Quality**
- Performance

Return values in synchronous massages are:

- Compulsory
- **May not used when response is obvious**
- Not used at all
- Represented by solid lines

According to Caper Jhones analysis of project activities, coding only has ------- affect part in system development.

- **13-14%**
- 36-40%
- 50-60%
- 70-80%

Normally a system is more easy to modify if its modules have

- High coupling and high cohesion
- High coupling and low cohesion
- Low coupling and high cohesion
- Low coupling and Low cohesion

https://vulmshelp.com/

AL-JUNAID INSTITUTE GROUP

In multi-threaded or multiprocessing applications where different execution threads may pass information to one another by sending ----------to each other.

https://vulmshelp.com/

- Interrupt calls
- Synchronous messages
- **Asynchronous messages**
- System Calls

Which of the following is not among the four layers of the object-oriented pyramid?

- The subsystem layers
- The class and object layer
- **The abstract layer**
- The message layers

System models include:

- User business processes
- User activities for conducting the business process
- Processes that need to be automated
- **All of the given options**

In the architecture trade-off analysis method the architectural style should be described using the

- Data flow view
- Module view
- Process view
- **All of the given**

---------- is concerned with decomposing the system into interacting sub-systems.

a) **System structuring**
b) Control Modelling
c) Molecular Decomposition
d) None of the given

A use case represents:

- A class, its attributes and operations
- An operation's interface and signature
- **The role a user plays when interacting with the system**
- The system's functionality for a particular purpose

External entity may be:

- Source of input data only
- **Source of input data and destination of results**
- Destination of results
- Repository of data

The process of utilizing our knowledge of computer science in effective production of ----------.

i. **Chemical Engineering**

https://vulmshelp.com/

AL-JUNAID INSTITUTE GROUP

ii. **Electrical Engineering**
iii. **Computer Engineering**
iv. **System Engineering**

Coupling is a measure of ---------- of a component.

- **Independence**
- Dependence
- Aggregation
- Composition

---------- has become a standard notation for object-oriented system modelling:

- **UML**
- C++
- OCL(Object Constraint Language)
- None of the given option

An arrow in data flow diagram represents:

- **Direction of flow of data**
- Processing of data
- External agent
- Internal Agent

---------- diagrams does not capture control flow information, it just shows the flow of data in a system.

- Sequence
- **Data Flow**
- Activity
- Class

In ---------- the analyst determines the source of requirements and where do these requirements consume:

- Data flow analysis
- **Source and sink analysis**
- Down parsing
- Up parsing

Data cannot flow from one external entity to other external entity because:

- It will get corrupted
- **It is not allowed in DFD**
- An external entity has no mechanism to read or write
- Both are outside the context of the system

In the functional design, the structure of the system resolves around:

- Objects

https://vulmshelp.com/

- Properties
- **Functions**
- All of the given options

---------- is one of the techniques to document domain knowledge

- **State transition diagram**
- Feasibility matrix
- System matrix
- None of the given options

In case of ---------- approach , decomposition of a problem revolves around data.

- **Object-Oriented**
- Action-Oriented
- Event-Oriented
- Process-Oriented

The ---------- relationship is a kind of a generalization specialization relationship:

- Bit-Byte
- Uses
- Binary
- **Extends**

Which statement is not true about system software?

- Change request has direct impact on software
- Passes through a constant process of evolution
- Change requests have direct impact in the form of defects
- None of the given

Strong cohesion implies that:

- **All parts of a component have a close logical relationship with each other**
- All parts of a component don't have a close relationship with each other
- Component is dynamic in nature
- Component is static in nature

The intent of Object-Oriented Analysis(OOA) is to define:

- All classes
- Relationships among classes
- Behaviour of classes
- **All of the given options**

Requirement engineering focuses on ---------- aspect of the software development process.

- Both what and how

https://vulmshelp.com/

AL-JUNAID INSTITUTE GROUP

- **What**
- How
- Why and how

---------- relationship is concerned with classes, not with class instantiates.

- Association
- **Inheritance**
- Aggregation
- Composition

Which of the following statements are true in context of the object model deviation through the Coad methodology?

a) A place is also a contains
b) Every container needs to be a place
c) Same person may play different times in the system.

- A only
- A and b
- **A and c**
- All of the given

The goal of ---------- is to translate the customer's desire for a set of defined capabilities into a working product.

- Electrical engineering
- **Product engineering**
- Hardware engineering
- Mechanical engineering

In case of a ------- message, the called routine that handles the message is completed before the caller resumes execution.

- Synchronous
- Asynchronous
- Bidirectional
- **A only**

ii. B only
iii. C only
iv. All of the given

In multiprocessing applications, different execution threads may pass information to one another by sending ------- to each other.

- Interrupt calls
- Synchronous messages
- **Asynchronous messages**
- System calls

https://vulmshelp.com/

AL-JUNAID INSTITUTE GROUP

A car is made up of a body, three or four wheels, a steering mechanism, a breaking mechanism, and a power engine"

The above statement is example of:

- **Whole-part relationship**
- Inheritance
- Specialization
- Generalization

To help separate an object's external behaviour from its implementation, the technique used is called -------

- Generalization
- Association
- Composition
- **Abstraction**

Sequences of messages can be present in:

(a) Use case diagram
(b) Sequence diagram
(c) Collaboration diagram

- a only
- b only
- c only
- **b and c**

Which of the following strategies lead to good software design:

- Separation of concerns
- Modularity
- Divide-and-conquer
- **All of the given options**

Data flow model:

- Captures the flow of data in a system
- Helps in developing an understanding of system's functionality
- Describes data origination, transformations and consumption in a system
- **All of the given options**

------- requirements are often called product features.

- Functional
- **Non-functional**
- Developer
- User

In the functional design, the structure of the system revolves around.

- Objects
- Properties

https://ulmshelp.com/

- **Functions**
- All of the given option

The first step in any OOA process model is to

- **Build an object-relationship model.**
- Define collaborations between objects.
- Elicit customer requirements
- Select a representation language

The ------ relationship is kind of a generalization specialization relationship.

- Bit-byte
- Uses
- Binary
- **Extends**

Regarding data flow model, which of the following statement(s) is true:

- It captures the transformation of data between processes/functions of a system
- **Processes on a data flow can operate in parallel**
- Only those processes are represented which we need to automate
- All of the given option

---------- is an extremely powerful technique for dealing with complexity.

- Aggregation
- **Abstraction**
- Inheritance
- Association

In “point of sale system”. the term “payment” represents

- Actor
- Participant
- **Transaction**
- Container

The architecture components for product engineering are

- **Data, hardware, software, people**
- Data, documentation, hardware, software
- Data, hardware, software, procedures
- Documentation, hardware, people, procedures

An object model encompasses the principle(s) of

- Abstraction
- Encapsulation
- Hierarchy or inheritance
- **All of the given option**

Prototyping is used when there is ---------- regarding requirements.

https://vulmshelp.com/

AL-JUNAID INSTITUTE GROUP

- **Uncertainty**
- Confirmation
- Conflict
- Consensus

In ------- phase of software development, requirement analyst focuses on possible design of the proposed solution.

- Maintenance
- Development
- **Definition**
- Vision

At which stage of software development loop, results are delivered?

- Problem definition
- Solution integration
- Technical development
- **Status quo**

A class will be cohesive if:

- Class does not implement complex interfaces
- Class does not have complex methods
- **If most of the methods do not use most of the data members most of the time**
- If most of the methods use most of the data members most of the time.

A DFD is normally levelled (adding more levels of abstraction) as

- It is a good idea in design
- It is recommended by many experts
- It is easy to do it
- **It is easier to read and understand a number of smaller DFDs than one large DFD**

Identify the true statement(s)

- An attribute that may have a number of values should be replaced by a new class and an object connection
- An attribute that varies over time, e.g. price of an item, should be replaced by an additional class with an affective data and value
- Replace “yes/no” type attribute with “status” type attributes for flexibility
- **All of given option**

---------- Is a technique in which we construct a model of an entity based upon its essential characteristics and ignore the inessential details.

- Inheritance

https://vulmshelp.com/

- Polymorphism
- Aggregation
- **Abstraction**

A structure is a manner of an organization which expresses a ------ strong organization within the problem domain.

- **Semantically**
- Syntactically
- Graphically
- None of the given

An object model of a system captures the ------ structure of a system.

- **Static**
- Dynamic
- Iterative
- Hierarchical

To determine the architectural style or combination of styles that best fits the proposed system, requirements engineering is used to uncover

- Algorithmic complexity
- **Characteristics and constraints**
- Control and data
- Design patterns

Which statement is not according to the software engineering principles? Software engineering is a(n) _____

- Balancing act
- Disciplined approach
- **Unsystematic approach**
- Quantifiable approach

In abbot's textual analysis technique, different part of speech is identified within the text of the specification and these part are modelled using different _____

- Event
- Process
- Operations
- **Components**

Quantitative methods for assessing the quality of proposed architectural designs are readily available.

- True
- **False**

In order to determine the role and responsibilities of the identified objects, we need to consider which of the following step(s):

https://vulmshelp.com/

AL-JUNAID INSTITUTE GROUP

a) Who I am?
b) What I know?
c) Who I know?
d) What I do?

- A only
- A and b
- B ,c and D
- **All of the given**

In object oriented design _____ layer contains the details that enable each object to communicates with its collaborators.

- Subsystem
- Responsibility
- **Message**
- Object

In sequence diagram, the boxes denote:

- **Objects (or classes)**
- Messages, sent from one object to other
- Life-time of objects
- None of the given option

In " railway tickit reservation system" the roles such as enquiry. Reservation and ticketing and cancellation are to be performed by the user called:

- **Passenger**
- System analyst
- System developer
- System designer

Class responsibilities are defined by _____

ii. Its attributes only
iii. Its collaborators
iv. Its operations only

iv. Both its attributes and operations

Requirement engineering mainly deals with the _____ of the system

- Vision phase
- **Definition phase**
- Development phase
- Maintenance phase

In UML based object oriented model of a system, a composition relation between two objects is shown by a _____ sign on the whole side of a relation line.

- An unfilled diamond

https://vulmshelp.com/

- A filled diamond
- A half diamond
- A dot

_____ analysis educates the analyst on business domain complexity and shows a way to deal with it.

- **Domain**
- Use case
- Object collaboration
- None of the given options

In this case of _____ intra component linkages are stronger while inter component linkages are weak.

- High cohesion
- Low coupling
- Low cohesion
- **High coupling**

An architectural style encompasses which of the following elements?

- Constraints
- Set of components
- Semantic models
- **All of the given**

In software engineering paradigm, any engineering approach must be founded on organizational commitment to _____

- Cost
- Scheduling
- **Quality**
- Performance

Identify the true statement:

- **Normally object oriented design is more maintainable than functional oriented.**
- Software with functional oriented design does not fulfil non functional requirements.
- Object oriented design can not implement “separation of concerns” strategy
- Function oriented design does not lead to an efficient product

A process in data flow diagram (DFD) represents

- Flow of data
- **Transformation of data**
- Storage of data
- An external agent

https://vulmshelp.com/

AL-JUNAID INSTITUTE GROUP

A maintainable design is a design , which supports

- Change
- Debugging
- Adding new features
- **All of the given**

Whole part structure is also called _____

- Generalization
- **Aggregation**
- Specialization
- Association

_____ are kind of umbrella activities that are used to smoothly and successfully perform the construction activities.

- Design activities
- **Management activities**
- Testing activities
- Maintenance activities

When you encounter both transform flow in the same DFD the flow is partitioned and the appropriate mapping technique is used on each part of the DFD.

- **True**
- False

Software architecture must address ---------- requirements of a software system.

- Functional
- Non-functional
- User Interface Requirements
- **Both functional and non-functional**.

To construct a system model the engineer should consider one of the following restraining factors.

- **Assumptions and constraints**
- Budget and expenses
- Data objects and operations
- Schedule and milestones

A cohesive class is one which emphasizes on ---------- unit of functionality.

- Single
- **Multiple**
- Static
- None

The best way to conduct a requirement validation review is to

https://vulmshelp.com/

- Examine the system model for errors
- have the customer look over the requirements
- Send them to the design team and see if they have any concerns
- **Use a checklist of the questions to examine each requirement**

Defining the services of an object means:

- What it does?
- **What it knows?**
- Who knows it?

Which one of the following is the external quality of a software product?

- Correctness
- Concision
- Cohesion
- **Low coupling**

In Data Flow Diagram, the entity or system, outside the boundary of this system is called:

- Process
- Data flow
- **External agent**
- Data store

GUI stands for:

- Genaric user Interface
- **Graphical user interface**
- Genaric user interaction
- Graphical user interaction

Specialization means:

- Calling the same method with the object of child object
- Hiding the data
- **Creating new subclasses for an existing class**
- None of the given options

In a use case diagram, an ellipse signifies a(n):

- Actor
- Class
- **Use case**
- System boundary

Software development is a step-by-step process, and in --------- phase of software development Business objective of an organization get cleared

- Maintenance
- Development

https://vulmshelp.com/

AL-JUNAID INSTITUTE GROUP

- Definition
- **Vision**

If you try to make software more user-friendly then the --------- may suffer.

- Reliability
- Software
- **Efficiency**
- Cost

In object-oriented design, the structure of the system revolves around.

- **Objects**
- Properties
- Methods
- All of the given option

In -------- relationship, a class shares the structure and behaviour defined in another class:

- Aggregation
- Composition
- **Inheritance**
- Uses

In Object Oriented Design, combining the services offered by an object with the attributes they work on, results in:

(a) Lower coupling and strong cohesion
(b) Lower cohesion and strong coupling
(c) Increased likelihood of reuse

(d) Decrease the modularity of the system

- A only
- B and c only
- **A and c only**

A change becomes ---------- because of close presence of data and functions.

- Accessible
- Global
- Private
- **Localized**

Software engineering is a ---------- approach.

- Systematic
- Disciplined
- Scheduled
- **All of the given options**

An external entity that interacts with the system is called a(n):

https://vulmshelp.com/

- Use case
- **Actor**
- Stakeholder
- Association

More powerful hardware resulted into the development of ----------powerful and --------- software.

- Less, complex
- **More , complex**
- More, simple
- Less, simple

A context diagram is used:

- As a first step in developing a detailed DFD of a system
- In systems analysis of very complex systems
- **As an aid to system design**
- As an aid to programmers

" System should maintain transaction log of every system"
The above statement is an example of:

- **Functional requirement**
- Non-functional requirement
- Pseudo requirement
- None of these

The architectural model provides the software engineer with the view of the system as a whole:

- **True**
- False

As per Peter Coad's methodology , which of the following may not be a perfect candidate for being an object?

- Zone
- Recipient
- Garage
- **Password**

In the case of ---------- approach , data is decomposed according to functionality requirements.

- Object-oriented
- **Action-oriented**
- Event-oriented
- Process-oriented

In UML based Object Oriented Model of a system, the diamond sign is used to depict ---------

https://vulmshelp.com/

- relations between two objects/classes.
  - Aggregation and Association
  - Inheritance and Association
  - **Composition and Aggregation**
  - Composition, Aggregation and Association

The system specification describes the:

- **Function and behavior of a computer-based system**
- Implementation of each allocated system element
- Algorithmic detail and data structures
- Time required for system simulation

In object oriented approach, ---------- are the people and organizations that take part in the system under consideration:

- **Actors**
- Places
- Participants

Software Design discusses ---------- aspect of software development.

- What
- **How**
- Who
- When

---------- requirements cause frequent modifications in user interface.

- Functional
- Non-functional
- **Unstable**
- User

By levelling a DFD (adding more levels of abstraction) we mean:

- **Splitting it into different levels**
- Make its structure uniform
- Expanding a process into one with more sub-processes giving more detail
- Summarizing a DFD to specify only these essentials

A "register" in "Point of Sale system" is an example of:

- Actor
- Participant
- **Tangible thing**
- Transaction

---------- is a set of processes and tools to develop software.

- **Software engineering**
- Information

https://vulmshelp.com/

- Software
- None of the given

The ---------- on which program operates is also considered as part of the software.

- **Data**
- Information
- Program
- None of the given

---------- diagram provides a time-based view and collaboration diagrams which provide an organization based view of the system's dynamics.

- Data flow diagram
- Entity relationship diagram
- Class diagram
- **Sequence diagram**

Synchronous messages are "call events" and are denoted by ----------

- **Full arrow**
- Half arrow
- <<create>>
- <<destroy>>

Which of the following are the components of system engineering software?

- Process
- Methods
- Tools
- **All of the given**

Identifying system features include ----------

- Log important information
- Conduct business
- Analyze business results
- **All of the above**

---------- is yet another technique that is used to reduce customer dissatisfaction at the requirement stage.

- Study of similar systems
- Site visits
- **Prototyping**
- All of the above

Data store notation in DFD represents:

- Data input
- Data output

https://vulmshelp.com/

AL-JUNAID INSTITUTE GROUP

- **Data input and data output**
- None of the above

The process of defining attributes is called ----------

- Who know me?
- **What I know?**
- Whom I know?
- All of the above

The output of the design process is a description of the:

- **Software architecture**
- Software Code
- Software
- All of the above

Which of the following are the levels of software requirements?

- Business requirements
- User requirements
- Functional requirements
- **All of the above**

Given below are some statements associated with data flow diagrams. Identify the correct statement among them.

- Data flow is made used of to model what systems do
- Flows of data can take place from a process to a sink
- **Context diagrams shows the major system processes**
- All processes have to be labelled or decomposed

In which of the following diagram the actors and attributes are represented with system boundary?

- Data flow diagram
- Entity relationship diagram
- Class diagram
- **Use case diagram**

_____ is real looking mock_up of what would be eventually delivered and might not do anything useful.

- Study of similar system
- Site visits
- **Prototyping**
- All of the above

_____ is blueprint for software construction.

- **Object oriented design**

https://vulmshelp.com/

- Sequence design
- Software design
- All of the above

_____ requirements lead to ill-spent time and rework.

- Unacceptable
- **Ambiguous**
- Dissatisfaction of customer
- None of the above

Which type of diagram is used to depict the dynamic behaviour of a system.

- ERD diagram
- DFD diagram
- Class diagram
- **Collaborations diagram**

What is the most crucial non-functional requirement of a system to control radiation dosages that are emitted as treatment for cancer?

- Security
- Reliability
- Easy usability
- **Accuracy**

A better design has an objective achieve

- High cohesion
- Low cohesion
- Low coupling
- **High cohesion and low coupling**

Which of the following are the components of software engineering framework is combine the three remaining components?

- Process
- Method
- **Tools**
- All of the above

In sequence diagrams the time required by the receiver object to process the message is denoted by an _____

- **Activation box**
- Message line
- Life line
- All of the above

How many types of OOD modes have _____.

https://vulmshelp.com/

AL-JUNAID INSTITUTE GROUP

- One
- **Two**
- Three
- Four

Which notation is used to represent the process of the system in DFD model.

- **Process**
- External agent
- Data flow
- Data store

Requirement engineering mainly deals with the _____ of the system

- Process
- Maintenance
- Development phase
- **Definition phase**

Insufficient user involvement leads to _____ products.

- **Unacceptable**
- Ambiguous
- Dissatisfaction of customer
- None of the above

Collaboration diagrams have basically two types of components: objects and _____

- Messages
- Method
- Classes
- **None of the above**

In object-oriented analysis how many number of tasks must occurs _____.

- 1
- **2**
- 3
- None of the above

State transition diagram is helpful in determining _____.

- Data store
- Process flow
- **Business understanding**
- None of the above

In sequence diagram events are organized in a _____ time life line.

- **Vertical**

https://vulmshelp.com/

- Horizontal
- Both A and B
- All of the above

Asynchronous messages are “signals,” denoted by _____.

- **Full arrow**
- Half arrow
- <<create>>
- <<destroy>>

When we write a program for computer and then we named it as _____ .

- Data
- Information
- **Software**
- None of the given

Context level diagram present in which of the following document.

- **SRS-software requirement specification**
- Design document
- Test phase
- All of the above

_____ is diagram in which objects are interact with each other and these are arranged in a sequence.

- ERD diagrams
- Inheritance diagrams
- Class diagrams
- **Sequence diagrams**

Which of the following layers are include in object-oriented design?

- The subsystem layers
- The class and object layer
- **All of the above**
- The message layers

Which notation is used to represent the boundary of the system in DFD model?

- Process
- **External agent**
- Data flow
- Data store

Identifying whole-part structures (aggregations) means what are my _____.

- **Components**
- Structures
- Class

https://vulmshelp.com/

- Object

An object or class may further be classified on the basis of _____ .

- Behaviour driven attributes
- Data driven attributes
- Responsibility driven attributes
- **All of the above**

DFD notation contains

- Process
- External agent
- Data flow
- **All of the above**

The doted lines in sequence diagram are called ____ .

- **Life line**
- Message line
- Entities line
- All of the above

An object may create another object via a _____ message .

- Full arrow
- half arrow
- **<<create>>**
- <<destroy>>

How many levels of software requirements are _____?

- One
- Two
- Three
- **Four**

Which of the following diagram has iterative activities?

- Data flow diagram
- Entity relationship diagram
- Class diagram
- **Use case diagram**

Which of the items listed below is not one of the software engineering layers?

- Tools
- **Manufacturing**
- Process
- Methods

Coupling is a measure of _____ a module or component.

https://vulmshelp.com/

AL-JUNAID INSTITUTE GROUP

- **Independent**
- Dependent
- Closeness
- All of the given

Software maintenance phase involves

- Debugging
- Adding new features
- Making changes
- **All of the given**

The hardest single part of building a software system is deciding precisely _____ to build.

- When
- **What**
- Why
- All of the given

Interaction diagrams depict the ____ behaviour of the system.

- Static
- Active
- **Dynamic**
- None of the given

A _____ can be used to describe the dynamic behaviour of an object-oriented system.

- ERD diagrams
- Inheritance diagrams
- Class diagrams
- **Series diagrams**

The Use case diagram shows that which ____ interact with each use case.

- Use case
- **Actor**
- Component
- Relation

Transactions are the _____ that must be remembered through time.

- **Events**
- Action
- Triggers
- Methods

A necessary supplement to transform or transaction mapping needed to create a complete architectural design is

- Entity relationship diagrams

https://vulmshelp.com/

AL-JUNAID INSTITUTE GROUP

- The data dictionary
- **Processing narratives for each module**
- Best cases for each module

i. ch module

https://vulmshelp.com/

VISIT ANY MORE FREELY SOLUTIONS

VUAnswer.com

CS504 QUIZ 1 SOLVED

Provide by VU Answer

1. Thin Client Model is one of the configurations of___Model.

a) Data-centered

b) client server

c) Layered

d) Reference

2. STL Stands for___

a) standard template library

b) Standard Type Link

c) Standard Tempo Line

d) Standard Type Link

3. The three basic principles that guide maintainability are:

Clarity

Enhancement

Flexibility

Simplicity

SUBSCRIBE OUR CHANNEL

https://www.youtube.com/channel/UCvFS8fAv4bvZsOmWuWMIg3Q

VISIT ANY MORE FREELY SOLUTIONS

VUAnswer.com

a) (a), (b) and (c)

b) (a), (c) and (d)

c) (b), (c) and (d)

d) All of above

4. Goto statements violate the idea of

a) object-oriented code

b) structured code

c) control structure

d) repetition structure

5. A function should not be larger in any case and should not exceed_____inlength

a) Half page

b) One page

c) Two pages

d) Three Pages

6. Data Centered Architectural Style is also called____

a) Repository model

b) Client Server model

c) Sub System model

SUBSCRIBE OUR CHANNEL

https://www.youtube.com/channel/UCvFS8fAv4bvZsOmWuWMIg3Q

VISIT ANY MORE FREELY SOLUTIONS

VUAnswer.com

d) Reference model

7. Global variables in C++ should always be referred to by using the

a) ::operator

b) :operator

c) Without an operator

d) None of the given

8. There are_layers in OSI reference model.

a) 5

b) 6

c) 7

d) 8

9. Fat Client Model is one of the configurations of____Model.

a) Data-centered

b) Layered

c) Reference

d) Client Server

10. Dynamic process model shows the process _____of the System.

SUBSCRIBE OUR CHANNEL

https://www.youtube.com/channel/UCvFS8fAv4bvZsOmWuWMIg3Q

VISIT ANY MORE FREELY SOLUTIONS

VUAnswer.com

a) Components

b) Objects

c) Structure

d) Linkage

11. Which of the following shows a commented statement in C++

a) # Ans = first + second

b) // Ans = first + second

c) \\ Ans = first + second

d) /# Ans = first + second

12. ___architectural model is widely used in Mainframe application.

a) Client-Server Model

b) Repository Model

c) Filter Model

d) Layered Model

13. Variables should be initialized where they are____and they should bedeclared in the___scope possible.

a) defined and smallest

b) declared and medium

SUBSCRIBE OUR CHANNEL

https://www.youtube.com/channel/UCvFS8fAv4bvZsOmWuWMIg3Q

VISIT ANY MORE FREELY SOLUTIONS

VUAnswer.com

c) defined and medium

d) declared and smallest

14. Identifier names also play a significant role in enhancing the____ of aprogram,

a) Writ ability

b) Readability

c) Reliability

d) All of the given choices

15.MVC stands for_

a) Model View Controller

b) Modern View Center

c) Model View Center

d) Modern View Controller

16. Which of the following is not a possible server in client server environment?

a) Database Server

b) Transaction Server

c) File Server

d) Time Server

SUBSCRIBE OUR CHANNEL

https://www.youtube.com/channel/UCvFS8fAv4bvZsOmWuWMIg3Q

VISIT ANY MORE FREELY SOLUTIONS

VUAnswer.com

17. In case of using unrelated operators in a single expression,____would be thebest choice to prevent the logical errors.

a) Comments

b) Indents

c) parenthesis

d) Short Cuts

18. In case of header files, construction is to avoid____ errors, the constructionshould appear in the top of the file (before the file header).

a) Compilation

b) run time

c) logical

d) All of the given

19. A self-documenting code is a code that explains itself without the need ofcomments and extraneous documentation, like____

a) Flowcharts

b) UML diagrams

c) Process-flow state diagrams

d) All of the given choices

20. Which one of these represents the Krutchen's 4+1 architectural view model

SUBSCRIBE OUR CHANNEL

https://www.youtube.com/channel/UCvFS8fAv4bvZsOmWuWMIg3Q

VISIT ANY MORE FREELY SOLUTIONS

VUAnswer.com

a) Logical view, Process view. Physical view. Development view. Usecase view

b) Logical view. Dynamic view. Physical view. Development view. Usecase view

c) Logical view. Process view, Physical view. Development view. Sequenceview

d) Dynamic view, Process view. Physical view, Development view. Usecase view

21. Client server is a_____system model.

a) Integrated

b) Distributed

c) Heterogeneous

d) Homogeneous

22. Vertical partitioning is also known as_____

a) Balancing

b) Mutating

c) Parallelizing

d) Factoring

23. "is" prefix should be used for_variables and methods.

a) General

b) Boolean

c) Constant

SUBSCRIBE OUR CHANNEL

https://www.youtube.com/channel/UCvFS8fAv4bvZsOmWuWMIg3Q

VISIT ANY MORE FREELY SOLUTIONS

VUAnswer.com

d) None of the given

24. Thin client model places a heavy processing load on_____

a) Only server

b) Only network

c) Both server and network

d) Neither server not network

25. C++ header files should have the extension. Source files can have theextension____

a) .c and .h

b) .cc and .h

c) **.h and .cpp**

d) All of the given

26. Which of following is among ten things, which the basic template of GOFdesign pattern includes.

a) IQ

b) context

c) Performance

d) Requirements

SUBSCRIBE OUR CHANNEL

https://www.youtube.com/channel/UCvFS8fAv4bvZsOmWuWMIg3Q

VISIT ANY MORE FREELY SOLUTIONS

VUAnswer.com

27. Vertical partitioning divides the architecture application from a__makingperspective.

a) decision

b) Design

c) Conclusion

d) Move

28. Issues like system performance, availability, scalability, and security areanalyzed in_

a) Logical View

b) physical view

c) Code View

d) Concurrency View

29. Class variables should never be declared public. Public variables violate whichof the following:

a) Information hiding

b) Encapsulation

c) Information hiding and Encapsulation

d) None of the give

30. Charles Simonyi first discussed Hungarian Notation, He was of_

a) Microsoft

SUBSCRIBE OUR CHANNEL

https://www.youtube.com/channel/UCvFS8fAv4bvZsOmWuWMIg3Q

VISIT ANY MORE FREELY SOLUTIONS

VUAnswer.com

b) IBM

c) Dell

d) Cisco

31. Patterns are devices that allow programs to share knowledge about their ___

a) Code

b) design

c) Analysis

d) Implementation

32. Process view in Krutchen's 4+1 architectural view model captures_____

a) Object model of the design

b) Concurrency and synchronization aspects of the design

c) Mapping(s) of the software onto the hardware and reflects its distributedaspect

d) Static organization of the software in its development environment

33. In the N-Tire architecture, the idea is to enhance scalability and_ by disturbing both the date and the application using multiple server machines

a) Usability

b) performance

c) Interoperability

SUBSCRIBE OUR CHANNEL

https://www.youtube.com/channel/UCvFS8fAv4bvZsOmWuWMIg3Q

VISIT ANY MORE FREELY SOLUTIONS

VUAnswer.com

d) Integrity

34. Names representing methods and functions should be___and written inmixed case starting with_case

a) Noun_____lower

b) Verb_____lower page

c) Noun_____upper

d) Verb_____upper

35. Floating point constant should always be written with decimal point and at

a) one decimal

b) Two decimals

c) Three decimals

d) none of the given

36. Static structural model shows the major system __

a) Leaks

b) Components

c) Activates

d) Objectives

SUBSCRIBE OUR CHANNEL

https://www.youtube.com/channel/UCvFS8fAv4bvZsOmWuWMIg3Q

VISIT ANY MORE FREELY SOLUTIONS

VUAnswer.com

37. Distributing the responsibilities to different subsystems so that we get a software system which is easy to maintain, is called_____ the architecture

a) Subtracting

b) partitioning

c) Cloning

d) Balancing

38. It ensures that a class only has one instance and provides a global point ofaccess to it

a) singleton pattern

b) Observer pattern

c) Real pattern

d) None of the given

39. N-tier architecture stems from the struggle to find a_____between the fat-client architecture and the thin client architecture

a) Concurrency

b) Distribution point

c) middle ground

d) Similarity

SUBSCRIBE OUR CHANNEL

https://www.youtube.com/channel/UCvFS8fAv4bvZsOmWuWMIg3Q

VISIT ANY MORE FREELY SOLUTIONS

VUAnswer.com

40. In client server model, the application is modeled as a set of___that areprovided by servers.

a) Request

b) Protocols

c) **service**

d) Requirements

41. Maintainability and portability issues are discussed in ___

a) Functional view

b) Physical view

c) code view

d) Logical view

42. In case of a file server, client request selected records from a___and the server transmits recorded to client over the network

a) Local memory

b) Network

c) Database

d) file

43. Physical view in Krutchen's 4+1 architectural view modelcaptures_____.

SUBSCRIBE OUR CHANNEL

https://www.youtube.com/channel/UCvFS8fAv4bvZsOmWuWMIg3Q

VISIT ANY MORE FREELY SOLUTIONS

VUAnswer.com

a) Object model of the design

b) Concurrency and synchronization aspects of the design

c) Mapping(s) of the software onto the hardware and reflectsitsdistributed aspect

d) Static organization of the software in its development environment

44.Logical view in Krutchen's 4+1 architectural view model captures.

a) Concurrency and synchronization aspects of the design

b) Mapping(s)of the software onto the hardware and reflects its distributedaspect

c) Object model of the design

d) Static organization of the software in its development environment

45. When programmers use shortcuts and cryptic codes in their program, _ becomes a major problem.

a) Maintenance

b) Reusability

c) Availability

d) Readability

46.Which one is correct?

a) double total = 0.5;

b) double total = .5;

SUBSCRIBE OUR CHANNEL

https://www.youtube.com/channel/UCvFS8fAv4bvZsOmWuWMIg3Q

VISIT ANY MORE FREELY SOLUTIONS

VUAnswer.com

c) double total = .50;

d) all of the given

47. “Description of communicating objects and classes that are customized to solvea general design problem in a particular context” is called ____.

a) Design Pattern

b) System Pattern

c) System Design

d) Design Method

48. Development view in Krutchen's 4+1 architectural view model captures

a) Object model of the design

b) Concurrency and synchronization aspects of the design

c) Mapping(s) of the software onto the hardware and reflects its distributedaspect

d) Static organization of the software in its development environment

49. Client server model tries to___ data and processing.

a) Distribute

b) Merge

c) Clone

d) Proceed

SUBSCRIBE OUR CHANNEL

https://www.youtube.com/channel/UCvFS8fAv4bvZsOmWuWMIg3Q

VISIT ANY MORE FREELY SOLUTIONS

VUAnswer.com

50. Which indent size eliminates the chance of code lines splitting?

a) 2

b) 4

c) 3

d) 6

51. Which one of these is a correct short cut used by the programmer?

a) x*=a

b) x=*a

c) x*a=

d) =a*x

52. ___ was the first pure Object-Oriented language in which observer patternwas used in implementing its Model View Controller pattern

a) Smalltalk

b) PASCAL

c) JAVA

d) C++

53. Identify the TRUE statement

SUBSCRIBE OUR CHANNEL

https://www.youtube.com/channel/UCvFS8fAv4bvZsOmWuWMIg3Q

VISIT ANY MORE FREELY SOLUTIONS

VUAnswer.com

a) Portability effects Security

b) Size of data types vary from one machine to other

c) Size of control structures vary from one machine to other

d) None of the given options

54. which of the following statements are same in output1) a = a >> 2

2) a = a/4

3) a = a * 2

a) (1) and (3) only

b) (2) and (3) only

c) (1) and (2) only

d) All produce the same result

56.Code should not be:

a) Commented

b) indented

c) cryptic

d) aligned

57. If (!(block < activeBlock))is equvivalent to:

a) if (block < activeBlock))

SUBSCRIBE OUR CHANNEL

https://www.youtube.com/channel/UCvFS8fAv4bvZsOmWuWMIg3Q

VISIT ANY MORE FREELY SOLUTIONS

VUAnswer.com

b) if ((block == activeBlock))

c) if ((block >= activeBlock))

d) None of the given

58. The form for (; ;) should be used for

a) Nested loop

b) Empty loop

c) More than 1000 iteration

d) Less than 1000 iterations

59. The clients and servers, in a client server environment, are connected through ____.

a) Framework

b) Interface

c) Middleware

d) Groupware

60. ___provides a unified interface to a set of intelligence in a sub-system

a) Observer pattern

b) Singleton pattern

c) Façade pattern

d) All of the above

SUBSCRIBE OUR CHANNEL

https://www.youtube.com/channel/UCvFS8fAv4bvZsOmWuWMIg3Q

VISIT ANY MORE FREELY SOLUTIONS

VUAnswer.com

61. Complex expressions:

a) Make the code easy to modify

b) Make the code difficult to modify

c) Make the code easy to understand

d) Does not affect understandability

62. Which of the following is used for multi-level commenting?

a) // Comment

b) /*Comment*/

c) (Comment)

d) */Comment/*

VU Answer
your right path

SUBSCRIBE OUR CHANNEL

https://www.youtube.com/channel/UCvFS8fAv4bvZsOmWuWMIg3Q

VISIT ANY MORE FREELY SOLUTIONS

VUAnswer.com

Note:

If you found any mistake in quiz correct on it.

Get more solved quizzes, assignment, past paper and vu solutions free to visit our website **VUAnswer.com.**

For help and suggestions contact us freely.

REGARD SARIM

WHATSAPP +923162965677

VU Answer
your right path

REMEMBER IN YOUR PRECIOUS PRAYERS

SUBSCRIBE OUR CHANNEL

https://www.youtube.com/channel/UCvFS8fAv4bvZsOmWuWMIg3Q

Ads by Google
Stop seeing this ad Why this ad?

CS504 Solved Quizzes File 1

Facebook Twitter



CS 504

SOLVED MCQ'S

Question # 1 of 10

Some bit field members are stored: I) left to right II) right to left III) in circular array

only (I) is true

Only (II) is true

Both (I) and (II) are true

All of the options (I, II and III) are true

Question # 2 of 10

Comma (,) is very dangerous because

Compiler does not recognise this symbol

It creates linkage problem

It causes side effects

All of the given options

Question # 3 of 10

_____ cause major portability issues

Loops

Bugs in code

Sizes of data types

Conditional Structures

Question # 4 of 10

In order to write a portable code which of the following guideline will be helpful:

Stick to the standards

Program in the mainstream

Size of data types

All of the given options

Question # 5 of 10

The complexity of a program may _____ if there are exceptional paths in it

Decrease

Increase

Remain same

All of given options

VU VICKY HOME PAST PAPERS HANDOUTS ASSIGNMENT SOLUTION GDB S

possible

STL

ANSI

ISO

CMMI

Question # 7 of 10

Exception handling provides:

Portability mechanism

Code Usability mechanism

Error Handling mechanism

All of the given options

Question # 8 of 10

Bit fields are a convenient way to express many difficult operations. However, bit fields suffer from one pr

Lack of usability

Lack of security

Lack of performance

Lack of portability

Question # 9 of 10

if (e.Title() == "CEO" || e.Salary() > 10000) in the above statement if e.Title()=="CEO") is TRUE then

Second part is not evaluated

Second part is always evaluated

Second part must also be TRUE

Second part is not evaluated but e.Salary() is called

Question # 10 of 10

A test case involves

Input/output specification plus a statement of the function under test

Steps to perform the function

Expected results that the software application produces

All of the given options

Client Server model tries to data and processing

Distribute

Merge

Clone

Proceed

Anti- Patterns is another concept that corresponds to common in analysis and design.

Mistake

Issues

Problems

All of the given

VU VICKY HOME PAST PAPERS HANDOUTS ASSIGNMENT SOLUTION GDB S

Three tier architecture contains layers

Presentation
Application
Database
All of the above

MVC stands for

Model View Controller
Modern View Center
Model View Center
Modern View Controller

Fat client model is one of the configurations of model

Data-Centered

Layered

Reference

Client Server

Description of communicating objects and classes that are customized to solved a general problem in a particular context is c

Design Pattern

System Patter

System Design

None of the Given

In the N-tire Architecture, the idea is to enhance scalability and by distributing both data and the application using multiple se

Usability

Performance

Interpretability

None of the given

Thin Client model is one of the configurations of ….. model

Data Centered

Client Server

Layered

Dynamic process model shows the process …….. of the system

Components

Objects

Structure

Linkage

It ensures that a class only has one instance and provides a global point of access to it.

Singleton Pattern

Observer Pattern

Real Pattern

None of the given

Question # 1 of 10 (Start time: 06:15:15 PM) Total Marks: 1

STL Stnads for ----------

Standard Template Library

Standard Type Link

Standard Tempo Line

None of the given

Question # 2 of 10 (Start time: 06:16:10 PM) Total Marks: 1

Three tier architecture contains ---------- layers

Select correct option:

Presentation

Application

Database

All of the above

Question # 3 of 10 (Start time: 06:17:05 PM) Total Marks: 1

Which of following is/are among ten things,which the basic template of GOF design pattern includes.

Select correct option:

Problem

VU VICKY HOME PAST PAPERS HANDOUTS ASSIGNMENT SOLUTION GDB S

All of the given

Question # 4 of 10 (Start time: 06:18:40 PM) Total Marks: 1

Vertical partitioning is also known as…..

Select correct option:

Balancing

Mutating

Parallelizing

Factoring

Question # 4 of 10 (Start time: 06:18:40 PM) Total Marks: 1

Vertical partitioning is also known as…..

Select correct option:

Balancing

Mutating

Parallelizing

Factoring

Question # 5 of 10 (Start time: 06:20:06 PM) Total Marks: 1

Patterns are devices that allow programs to share knowledge about their ----------.

Select correct option:

Code

Design

Analysis

None of the given

Question # 6 of 10 (Start time: 06:22:07 PM) Total Marks: 1

Client server model tries to …….data and processing.

Select correct option:

Distribute

VU VICKY HOME PAST PAPERS HANDOUTS ASSIGNMENT SOLUTION GDB S

Proceed

Question # 7 of 10 (Start time: 06:23:24 PM) Total Marks: 1

N-tier architecture stems from the struggle to find a ---------- between the fat-client architecture and the thin-client architectu

Select correct option:

Concurrency

Distribution point

Middle ground

Similarity

Question # 8 of 10 (Start time: 06:24:20 PM) Total Marks: 1

Vertical partitioning divides the architecture application from a …… making perspective.

Decision

Design

Conclusion

Move

Question # 9 of 10 (Start time: 06:25:23 PM) Total Marks: 1

Distributing the responsibilities to different subsystems so that we get a software system which is easy to maintain, is called . architecture.

Select correct option:

Subtracting

Partitioning

Cloning

Balancing

Question # 1 of 10 (Start time: 01:35:10 PM) Total Marks: 1

The nominal case should be put in the if-part and the exception in the else-part of an if statement.

Select correct option:

TURE

FALSE

VU VICKY HOME PAST PAPERS HANDOUTS ASSIGNMENT SOLUTION GDB S

Charles Simonyi first discussed Hungarian Notation. He was of ------ .

Select correct option:

Microsoft

IBM

Dell

Cisco

sec(s)

Question # 3 of 10 (Start time: 01:36:25 PM) Total Marks: 1

The terms get/set must be used where an attribute is accessed

Select correct option:

Indirectly

Directly

sec(s)

Question # 4 of 10 (Start time: 01:36:37 PM) Total Marks: 1

A self documented program/code contains the following attribute(s):

Size of each function

Choice of variable

Choice of variable

All of the given choices

sec(s)

Question # 5 of 10 (Start time: 01:37:42 PM) Total Marks: 1

"is" prefix should be used for------ variables and methods.

Select correct option:

General

Boolean

Constant

None of the given

sec(s)

Question # 6 of 10 (Start time: 01:38:16 PM) Total Marks: 1

The use of comments should be minimized by making the code self-documenting by appropriate name choices and an explicit logical structure.

Select correct option:

TRUE

VU VICKY HOME PAST PAPERS HANDOUTS ASSIGNMENT SOLUTION GDB S

Question # 7 of 10 (Start time: 01:38:44 PM) Total Marks: 1

Variables should be initialized where they are ------and they should be declared in the ------scope possible.

Select correct option:

defined and smallest

declared and medium

defined and medium

declared and smallest

sec(s)

Question # 8 of 10 (Start time: 01:39:56 PM) Total Marks: 1

Unrelated variables should be declared in the same statement.

Select correct option:

True

False

sec(s)

Question # 9 of 10 (Start time: 01:40:22 PM) Total Marks: 1

which of the following statements are same in output: 1) a = a >> 2 2) a = a / 4 3) a = a * 2

Select correct option:

(1) and (3) only

(2) and (3) only

(1) and (2) only

All procduce the same result

sec(s)

Question # 10 of 10 (Start time: 01:41:38 PM) Total Marks: 1

Goto statements violate the idea of

Select correct option:

object oriented code

structured code

control structure

repetition structure

sec(s)

Question # 2 of 10 (Start time: 01:44:29 PM) Total Marks: 1

MVC pattern was based on the ---------- pattern.

Select correct option:

VU VICKY HOME PAST PAPERS HANDOUTS ASSIGNMENT SOLUTION GDB S

Behavioral

None of given

sec(s)

Question # 3 of 10 (Start time: 01:44:56 PM) Total Marks: 1

Which one is correct?

Select correct option:

double total = 0.5;

double total = .5;

double total = .50;

all of the given

sec(s)

Question # 4 of 10 (Start time: 01:46:09 PM) Total Marks: 1

Code should not be:

Select correct option:

commented

indented

cryptic

aligned

sec(s)

Question # 5 of 10 (Start time: 01:46:41 PM) Total Marks: 1

Global variables in C++ should always be referred to by using the

Select correct option:

:: operator

: operator

Without an operator

None of the given

sec(s)

Question # 6 of 10 (Start time: 01:47:17 PM) Total Marks: 1

_____ was the first pure Object Oriented language in which observer pattern was used in implementing its Model View Controller pattern

Select correct option:

Smalltalk

PASCAL

VU VICKY HOME PAST PAPERS HANDOUTS ASSIGNMENT SOLUTION GDB S

sec(s)

Question # 7 of 10 (Start time: 01:47:56 PM) Total Marks: 1

using proper paranthesis normally makes the code

Select correct option:

easy to read

easy to understand

less ambigous

All of the given options

sec(s)

Question # 9 of 10 (Start time: 01:48:58 PM) Total Marks: 1

Which of the following shows a commented statement in C++

Select correct option:

Ans = first + second

// Ans = first + second

\\ Ans = first + second

/# Ans = first + second

sec(s)

Question # 1 of 10 (Start time: 01:51:42 PM) Total Marks: 1

The form for (;;) should be used for

Select correct option:

nested loop

empty loop

more than 1000 iterations

sec(s)

sec(s)

Question # 2 of 10 (Start time: 01:52:09 PM) Total Marks: 1

Identifier names also play a significant role in enhancing the -------- of a program.

Select correct option:

Writ ability

Readability

Reliability

sec(s)

Question # 3 of 10 (Start time: 01:53:08 PM) Total Marks: 1

VU VICKY HOME PAST PAPERS HANDOUTS ASSIGNMENT SOLUTION GDB S

Singleton Pattern

Observer Pattern

Real Pattern

None of the given

sec(s)

Question # 4 of 10 (Start time: 01:53:49 PM) Total Marks: 1

Names representing methods and functions should be----and written in mixed case starting with -----case.

Select correct option:

Noun—lower

Verb----lower

Noun ---upper

Noun----upper

sec(s)

Question # 5 of 10 (Start time: 01:54:45 PM) Total Marks: 1

A function should not be larger in any case and should not exceed _____ in length .

Select correct option:

Half page

One page

Two pages

Three Pages

sec(s)

Question # 7 of 10 (Start time: 01:55:56 PM) Total Marks: 1

-----provides a unified interface to a set of interfaces in a sub-system.

Select correct option:

Observer Pattern

Singleton Pattern

Façade Pattern

All of the above

sec(s)

Question # 8 of 10 (Start time: 01:56:35 PM) Total Marks: 1

MVC stands for ----------

VU VICKY HOME PAST PAPERS HANDOUTS ASSIGNMENT SOLUTION GDB S

Modern View Center

Model View Center

Modern View Controller

sec(s)

Question # 8 of 10 (Start time: 01:59:40 PM) Total Marks: 1

A self documenting code is a code that explains itself without the need of comments and extraneous documentation, like _____

Select correct option:

Flowcharts

UML diagrams

Process-flow state diagrams

All of the given choices

sec(s)

Question # 10 of 10 (Start time: 02:00:38 PM) Total Marks: 1

Complex expressions:

Select correct option:

Make the code easy to modify

Make the code difficult to modify

Make the code easy to understand

Does not effect understandablity

Question # 10 of 10 (Start time: 01:49:27 PM) Total Marks: 1

Comments should NOT be indented relative to their position in the code

Select correct option:

TRUE

FALSE

----Provides a unified interface to a set of interfaces in a sub-system

Observer Pattern

Singleton Pattern

Façade Pattern

All of the above

VU VICKY HOME PAST PAPERS HANDOUTS ASSIGNMENT SOLUTION GDB S

Application

Protocol

Services

All of the above

Vertical Partitioning is also know as

Balancing

Mutating

Parallizing

Factoring

Façade Pattern provides a unified interface to a set of interfaces ina sub-system.

True

False

Which of the is/are among ten things which the basis template of GOF design pattern includes.

Problem

Context

Forces

All of the above

Thin Client Model places a heavy processing load on……

Only Server

Only Network

Both Server and Network(Not Sure)

Neither server nor network

Zero install architecture does not need any installation on _____.

VU VICKY HOME PAST PAPERS HANDOUTS ASSIGNMENT SOLUTION GDB S

Client & Server Side (Not Sure)

None of the above

Data-Centered Architectural Style is also called ….

Repository model

Client Server model

Sub system model

Reference model

Client Server model tries to data and processing.

Distribute
Merge
Clone
Proceed

Ads by Google
Stop seeing this ad Why this ad?

Tags solved quiz Solved Quizzes CS504

Facebook Twitter

You might like



CS301 Solved Quizzes File 1



CS504 Solved Quizzes File 3



CS504 Solved Quizzes

0 Comments



Add a comment...

Facebook Comments plugin

‹ Previous Post

VU VICKY

ABOUT US

Welcome to VU Vicky. If you need assistance whatsappp me at 03037155010

Copyright © 2018-2021 by - VU Vicky

Ads by Google
Stop seeing this ad Why this ad?

CS504 Solved Quizzes File 2

Facebook Twitter

Ads by Google
Stop seeing this ad Why this ad?

SOLVED MCQ'S

Client Server model tries to data and processing.

Distribute
Merge
Clone
Proceed

Anti- Patterns is another concept that corresponds to common in analysis and design.
Mistake
Issues
Problems
All of the given

Three tier architecture contains layers.
Presentation
Application
Database

All of the above

MVC stands for
Model View Controller
Modern View Center
Model View Center
Modern View Controller

Fat client model is one of the configurations of model.
Data-Centered
Layered
Reference
Client Server

Description of communicating objects and classes that are customized to solved a general problem in a p called.
Design Pattern
System Patter

In the N-tire Architecture, the idea is to enhance scalability and by distributing both data and the applicati server machines.

Usability

Performance

Interpretability

None of the given

Thin Client model is one of the configurations of ….. model.

Data Centered

Client Server

Layered

Reference

Dynamic process model shows the process …….. of the system.

Components

Objects

Structure

Linkage

It ensures that a class only has one instance and provides a global point of access to it.

Singleton Pattern

Observer Pattern

Real Pattern

None of the given

STL Stnads for ---------- .

Standard Template Library

Standard Type Link

Standard Tempo Line

None of the given

Three tier architecture contains ---------- layers.

Presentation

Application

Database

All of the above

Which of following is/are among ten things, which the basic template of GOF design pattern includ

Problem

All of the given

Vertical partitioning is also known as….. .

Balancing
Mutating
Parallelizing
Factoring

Vertical partitioning is also known as…..
Balancing
Mutating
Parallelizing
Factoring

Patterns are devices that allow programs to share knowledge about their -------.
Code
Design
Analysis
None of the given

Client server model tries to …….data and processing.
Distribute
Merge
Clone
Proceed

N-tier architecture stems from the struggle to find a ---------- between the fat-client architecture and the thi architecture.
Concurrency
Distribution point
Middle ground
Similarity

Vertical partitioning divides the architecture application from a …… making perspective.
Decision
Design
Conclusion
Move

Subtracting
Partitioning
Cloning
Balancing

The nominal case should be put in the if-part and the exception in the else-part of an if statement.
TURE
FALSE

Charles Simonyi first discussed Hungarian Notation. He was of ------. .
Microsoft
IBM
Dell
Cisco

The terms get/set must be used where an attribute is accessed.
Indirectly
Directly

A self documented program/code contains the following attribute(s):
Size of each function
Choice of variable
Choice of variable
All of the given choices

"is" prefix should be used for------ variables and methods.
General
Boolean
Constant
None of the given

The use of comments should be minimized by making the code self-documenting by appropriate name ch
explicit logical structure.
TRUE
FALSE

Variables should be initialized where they are ------and they should be declared in the ------scope possible.
defined and smallest
declared and medium
defined and medium
declared and smallest

True
False

which of the following statements are same in output: 1) a = a >> 2 2) a = a / 4 3) a = a * 2.
(1) and (3) only
(2) and (3) only
(1) and (2) only
All produce the same result

Goto statements violate the idea of.
object oriented code
structured code
control structure
repetition structure

MVC pattern was based on the ---------- pattern.
Observer
Structural
Behavioral
None of given

Which one is correct?
double total = 0.5;
double total = .5;
double total = .50;
all of the given

Code should not be:
commented
indented
cryptic
aligned

Global variables in C++ should always be referred to by using the.
:: operator
: operator
Without an operator
None of the given

_____ was the first pure Object Oriented language in which observer pattern was used in impl nti
View Controller pattern

JAVA
C++

Using proper paranthesis normally makes the code.
easy to read
easy to understand
less ambigous
All of the given options

Which of the following shows a commented statement in C++.
Ans = first + second
// Ans = first + second
\\ Ans = first + second
/# Ans = first + second

The form for (;;) should be used for.
nested loop
empty loop
more than 1000 iterations

Identifier names also play a significant role in enhancing the -------- of a program.
Writ ability
Readability
Reliability

It ensures that a class only has one instance and provides a global point of access to it.
Singleton Pattern
Observer Pattern
Real Pattern
None of the given

Names representing methods and functions should be----and written in mixed case starting
with -----case.
Noun—lower
Verb----lower
Noun ---upper

A function should not be larger in any case and should not exceed _____ in length .
Half page
One page
Two pages

-----provides a unified interface to a set of interfaces in a sub-system.

Observer Pattern

Singleton Pattern

Façade Pattern

All of the above

MVC stands for ----------.

Model View Controller

Modern View Center

Model View Center

Modern View Controller

A self documenting code is a code that explains itself without the need of comments and extraneous documentation, like _____.

Flowcharts

UML diagrams

Process-flow state diagrams

All of the given choices

Complex expressions:

Make the code easy to modify

Make the code difficult to modify

Make the code easy to understand

Does not effect understandability

Comments should NOT be indented relative to their position in the code.

TRUE

FALSE

----Provides a unified interface to a set of interfaces in a sub-system.

Observer Pattern

Singleton Pattern

Façade Pattern

All of the above

In Client Server model, there application is modeled as a set so --- that are provided by server.

Application

Protocol

Services

All of the above

Balancing
Mutating
Parallizing
Factoring

Façade Pattern provides a unified interface to a set of interfaces ina sub-system.
True
False

Which of the is/are among ten things which the basis template of GOF design pattern includes.
Problem
Context
Forces
All of the above

Thin Client Model places a heavy processing load on…….
Only Server
Only Network
Both Server and Network(Not Sure)
Neither server nor network

Zero install architecture does not need any installation on _____.
Server side (not Sure)
Client side
Client & Server Side (Not Sure)
None of the above

Data-Centered Architectural Style is also called …..
Repository model
Client Server model
Sub system model
Reference model

Stop seeing this ad Why this ad?

Str = 0;// str is string Which rewritten from of above line of code is more in line with the self-documentatic the code above.
Str = false;
Str = NULL;
Str = '\0;
Str = 0.0;
Struct packed_struct { unsigned int f1:1;} pack; Here in "packed_struct":
Value of f1 =1
Size of f1 = 1 bit
Value of f1 should not exceede 1 (Not Sure)
None of given options

Ads by Google
Stop seeing this ad Why this ad?

In the switch statement, cases should always end with a -------statment.
Switch
Go
Break
Stop

Bit fields are a convenient way to express many difficult operations. However, bit fields suffer from one pr
Lack of Usability
Lack of Security
Lack of Performance
Lack of Portability

If ((block >= activeBlock))
None of the given

Ads by Google
Stop seeing this ad Why this ad?

Bit fields allow the packing of data in a structure. Using Bit fields we can.
Read 9 bit integers
Avoid memory leakages
Avoid memory overflow
Avoid syntax errors

The order in which bytes of one word are stored is _____ dependent.
Hardware
Software
Language
Syntax

In order to write a portable code which of the following guideline will be helpful:
Stick to the standards
Program in the mainstream
Size of data types
All of give options

Complex expressions.
Make the code easy to modify
Make the code difficult to modify
Make the code easy to understand
Does not effect understandability

In the switch statement, cases should always end with a -------statement.
Switch
Go
Break
Stop

x = 0; // x is floating pt Which rewritten form of above line of code is more in line with the self-documentat than the code above.
x = false

using proper paranthesis normally makes the code.
easy to read
easy to understand
less ambiguous
All of the given options

The use of comments should be minimized by making the code self-documenting by appropriate name ch explicit logical structure.
True
False

Bit fields allow the packing of data in a structure. using Bit fields we can:
Read 9 bit integers
Avoid memory leakages
Avoid memory overflow
Avoid syntax errors

1) x = (a + 2 > 3)? a : a-1 ; 2) if((a + 2)> 3) x = a; else x = a - 1;
Statement (2) is more complex than (1)
Statement (2) is more complex than (1)
Both statements are very complex
None of the given option

80/20 rule states that:
you spend 80 percent of your time in 20 percent of the code
you spend 20 percent of your time in 80 percent of the code
We should try to optimized 80 percent or at least 20 percent of the code
None of the given options.

Be very careful when you use functions with side effects – functions that change the values of the _____
Objects
Classes
Structures
Variables

Comma (,) is very dangerous because.
Compiler does not recognise this symbol
It creates linkage problem
It causes side effects
All of the given options (Not Sure)

The C/C++ language has not specified whether ____ is arithmetic or logical.
Right shift >>
Right shift <<
&&
||

In order to make a code more portable, Instead of using vendor specific language extensions, use __ possible.

CMMI

The order in which bytes of one word are stored is _____ dependent.
hardware
software
language
syntax

Be very careful when you use functions with side effects – functions that change the values of the _____
Objects
Classes
Structures
Variables

When a small set of functions (which use each other) is so overwhelmingly the bottleneck, there are two a
use a better algorithm OR re-write the code
debug the code OR place assertions in code
remove the functions OR add more functions
changed programming language OR compiler at least

_____ cause major portability issues
Loops (Not Sure)
Bugs in code
Sizes of data types
Conditional Structures

In the switch statement, cases should always end with a -------statment.
Switch
Go
Break
Stop

Some bit field members are stored: I) left to right II) right to left III) in circular array.
only (I) is true
Only (II) is true
Both (I) and (II) are true
All of the options (I, II and III) are true

x = 0; // x is floating pt Which rewritten form of above line of code is more in line with the self-documentat than the code above.
x = false
x = NULL
x = 0.0;
x = '\0'

It causes side effects
All of the given options(Not sure)

using proper paranthesis normally makes the code
easy to read
easy to understand
less ambiguous
All of the given options

The use of comments should be minimized by making the code self-documenting by appropriate name ch
explicit logical structure.
True
False

Bit fields allow the packing of data in a structure. using Bit fields we can:
Read 9 bit integers
avoid memory leakages
avoid memory overflow
Avoid syntax errors

Be very careful when you use functions with side effects – functions that change the values of the _____
Objects
Classes
Structures
Variables

A test case involves
Input/output specification plus a statement of the function under test
Steps to perform the function
Expected results that the software application produces
All of the given options

The C/C++ language has not specified whether ____ is arithmetic or logical.
Right shift >>
Right shift <<
&&
||

If an application fulfills its specifications but deviates from users expectations or their desired behavior. T
software is verified but not ----------
Validated
Corrected
Checked
Traced

The raising of the imaginary error flag is simply called raising or _____ an error.
Catching

struct packed_struct { unsigned int f1:1; } pack; Here in "packed_struct":
value of f1 = 1
size of f1 = 1 bit
value of f1 should not exceede 1
None of given options

The idea behind exception handling is to raise some error flag every time _____.
The code compiles
The code links
Memory is allocated
Something goes wrong

Bit fields allow the packing of data in a structure. using Bit fields we can:
Read 9 bit integers
avoid memory leakages
avoid memory overflow
Avoid syntax errors

The order in which bytes of one word are stored is _____ dependent.
hardware
software
language
syntax

A _____ is a variance from a desired product attribute.
Exception
Error
Mistake
Defect

The C/C++ language does not define the alignment of items within.
structures
classes
unions
All of the given options

The raising of the imaginary error flag is simply called raising or _____ an error
Catching
Casting
Throwing
None of given options

Consider the following statement: int a,b=10; Which of the following is correct:
variable "a" is initialized to 10
Variable "b" is initialized to 10

Exception handling is a powerful technique that separates error-handling code from _____ code.
Normal
Faulty
Buggy
Complex

Bit fields are a convenient way to express many difficult operations. However, bit fields suffer from one pr
Lack of usability
Lack of security
Lack of performance
Lack of portability

The complexity of a program may _____ if there are exceptional paths in it
Decrease
Increase
Remain same
All of given options

When an error is thrown the overall system (on the lookout for this error flag) responds by _____ the erro
Ignoring
Casting
Catching
All of the given options(Not Sure)

Comments are not syntax checked
TRUE
FALSE

Modularity is a tool that can help us in increasing the size of individual functions, making them less reada
True
False

Tags solved quiz Solved Quizzes CS504

Facebook Twitter

You might like



CS301 Solved Quizzes File 1



CS504 Solved Quizzes File 3



CS504 Solved Quizzes



VU VICKY

ABOUT US

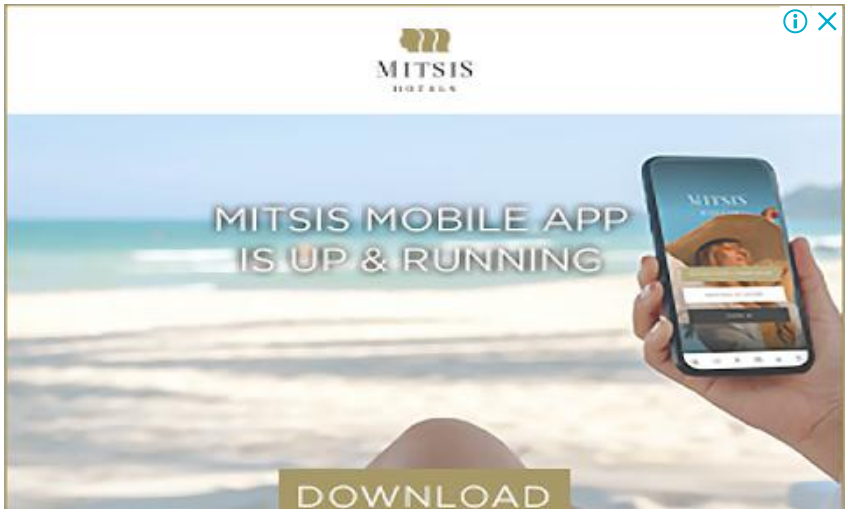
Welcome to VU Vicky. If you need assistance whatsappp me at 03037155010

Copyright © 2018-2021 by - VU Vicky

Ads by Google
Stop seeing this ad Why this ad?

CS504 Solved Quizzes File 3

Facebook Twitter



CS 504

SOLVED MCQ'S

Question # 1 of 10 (Start time: 05:08:46 PM) Total Marks: 1

The pattern movement became very quiet until ---------- when patterns appeared again at an conference.

Select correct option:

- **1987**

1988

1962

1995

Question # 2 of 10 (Start time: 05:09:52 PM) Total Marks: 1

Which piece of code is self-documented?

Select correct option:

if (x==0) // this is the case when we are allocating a new number

if (AllocFlag == 0)

- **If (AllocFlag == NEW_NUMBER)**

None of the given

Question # 3 of 10 (Start time: 05:11:13 PM) Total Marks: 1

Variables should be initialized where they are ------and they should be declared in the ------scop

Select correct option:

defined and smallest

defined and medium

- declared and smallest

Click here to Save Answ er & Move to Next Question

Question # 4 of 10 (Start time: 05:11:55 PM) Total Marks: 1

Complex expressions should be broken down into multiple statements. Select the

Ads by Google
Stop seeing this ad Why this ad?

Correct one *x += (*xp= (2*k < (n-m)? c [k+1]: d [k--]));

Select correct option:

- if (2*k < n-m) *xp = c[k+1]; else *xp = d[k--]; *x = *x + *xp;

if (2*k <= n-m) *xp = c[k+1]; *xp = d[k--]; *x = *x + *xp;

if (2*k < n-m) *xp = d[k--]; else *xp = c[k+1]; *x = *x + *xp;

None of the given

Question # 5 of 10 (Start time: 05:13:12 PM) Total Marks: 1

Goto statements violates the idea of

Select correct option:

object oriented code

- Structured code

control structure

Repetition structure

In the case of zero-install, the network environment is used to ---------server side processing by a of servers which share processing load.

Select correct option:

- **Distribute**

Centeralize

Shift

None of the above

Question # 7 of 10 (Start time: 05:15:14 PM) Total Marks: 1

Which one is the most desirable quality of a software artifact?

Select correct option:

Reliability

Performance

- **Maintainability**

All of the given

Question # 8 of 10 (Start time: 05:16:37 PM) Total Marks: 1

Which of these is a graphical notation for depicting procedural detail?

Select correct option:

Box diagram

- **Decision table**

ER diagram

Graph matrix

Question # 9 of 10 (Start time: 05:18:25 PM) Total Marks: 1

A decision table should be used

Select correct option:

To guide the development of the project management plan

only when building an expert system

When a complex set of conditions and actions appears in a component

Question # 10 of 10 (Start time: 05:19:09 PM) Total Marks: 1

Ads by Google
Stop seeing this ad Why this ad?

N-tier architecture stems from the struggle to find a --------between the fat client

architecture and the thin-client architecture.

Select correct option:

Concurrency

Concurrency

- **Middle ground**

The nominal case should be put in the if-part and the exception in the else-part of an if stateme

Select correct option:

- **TURE**

FALSE

Question # 2 of 10 (Start time: 01:35:59 PM) Total Marks: 1
Charles Simonyi first discussed Hungarian Notation. He was of ----- .

Select correct option:

- **Microsoft**

Dell

Cisco

sec(s)

Question # 3 of 10 (Start time: 01:36:25 PM) Total Marks: 1
The terms get/set must be used where an attribute is accessed

Select correct option:
Indirectly

- **Directly**

sec(s)

Question # 4 of 10 (Start time: 01:36:37 PM) Total Marks: 1
A self documented program/code contains the following attribute(s):

Size of each function

Choice of variable

Choice of variable

- **All of the given choices**

sec(s)

Question # 7 of 10 (Start time: 01:55:56 PM) Total Marks: 1
-----provides a unified interface to a set of interfaces in a sub-system.

Select correct option:
- **Observer Pattern**

Singleton Pattern

Façade Pattern

All of the above

sec(s)

Question # 8 of 10 (Start time: 01:56:35 PM) Total Marks: 1
MVC stands for ----------

Select correct option:

Model View Center

Modern View Controller

sec(s)

Question # 8 of 10 (Start time: 01:59:40 PM) Total Marks: 1
A self documenting code is a code that explains itself without the need

of comments and

extraneous documentation, like _____

Select correct option:
Flowcharts

UML diagrams

Process-flow state diagrams

- **All of the given choices**

sec(s)

Question # 10 of 10 (Start time: 02:00:38 PM) Total Marks: 1
Complex expressions:

Select correct option:
Make the code easy to modify

- **Make the code difficult to modify**

Make the code easy to understand

Does not effect understandablity

Question # 10 of 10 (Start time: 01:49:27 PM) Total Marks: 1
Comments should NOT be indented relative to their position in the

code

Select correct option:
TRUE

- **FALSE**

Select correct option:
double total = 0.5;
double total = .5;
double total = .50;
all of the given

Question # 2 of 10 (Start time: 11:19:21 PM) Total Marks: 1
Be very careful when you use functions with side effects – functions that change the values of the __
Select correct option:
Objects
Classes
Structures
Variables

Question # 3 of 10 (Start time: 11:20:10 PM) Total Marks: 1
_____ is a tool that can help us in reducing the size of individual functions.
Select correct option:
Inheritance
Modularity
Association
Abstraction

Question # 4 of 10 (Start time: 11:20:38 PM) Total Marks: 1
The expressions with logical operators can be evaluated only from _____.
Select correct option:
Right to left
Left to right
Top to bottom
Bottom to top

Question # 5 of 10 (Start time: 11:21:06 PM) Total Marks: 1
80/20 rule states that:
Select correct option:
you spend 80 percent of your time in 20 percent of the code
you spend 20 percent of your time in 80 percent of the code
We should try to optimized 80 percent or at least 20 percent of the code
None of the given options.

Question # 6 of 10 (Start time: 11:21:30 PM) Total Marks: 1

Observer Pattern
Singleton Pattern
Façade Pattern
All of the above

Question # 7 of 10 (Start time: 11:22:02 PM) Total Marks: 1
Which of following is/are among ten things,which the basic template of GOF design pattern includes
Select correct option:
Problem
Context
Forces
All of the given

Question # 8 of 10 (Start time: 11:22:29 PM) Total Marks: 1
_____ and _____ are two important tools that helps in managing the program complexity.
Select correct option:
Composition, Inheritance
Abstraction, Encapsulation
Aggregation, Inheritance
Modularity, Composition

Question # 9 of 10 (Start time: 11:22:55 PM) Total Marks: 1
Comma (,) is very dangerous because
Select correct option:
Compiler does not recognise this symbol
It creates linkage problem
It causes side effects
All of the given options

Question # 10 of 10 (Start time: 11:23:20 PM) Total Marks: 1
Goto statements violate the idea of
Select correct option:
object oriented code
structured code
control structure
repetition structure

Question # 1 of 10 (Start time: 10:41:25 PM) Total Marks: 1
If a function changes the value of some other accessible data object along with returning its va aft
called _____.

Modularity
Side Effects
Abstraction

Question # 2 of 10 (Start time: 10:41:47 PM) Total Marks: 1
if a major rework is required to translate a program written for one environment to another, it means
_____
Select correct option:
Secure
Bug Free
plateform dependent
Portable

Question # 3 of 10 (Start time: 10:42:21 PM) Total Marks: 1
It ensures that a class only has one instance and provides a global point of access to it.
Select correct option:
Singleton Pattern
Observer Pattern
Real Pattern
None of the given

Question # 4 of 10 (Start time: 10:42:55 PM) Total Marks: 1
Code should not be:
Select correct option:
commented
indented
cryptic
aligned

Question # 5 of 10 (Start time: 10:43:16 PM) Total Marks: 1
In case of header files, construction is to avoid ------ errors. The construction should appear in the top
the file header).
Select correct option:
compilation
run time
logical
All of the given

Question # 6 of 10 (Start time: 10:43:46 PM) Total Marks: 1

function
object
data type
none of the above

Question # 7 of 10 (Start time: 10:44:09 PM) Total Marks: 1
Identifier names also play a significant role in enhancing the -------- of a program.
Select correct option:
Writ ability
Readability
Reliability
All of the given choices

Question # 8 of 10 (Start time: 10:44:39 PM) Total Marks: 1
A function should not be larger in any case and should not exceed _____ in length .
Select correct option:
Half page
One page
Two pages
Three Pages

Question # 9 of 10 (Start time: 10:45:00 PM) Total Marks: 1
_____ was the first pure Object Oriented language in which observer pattern was used in impler
View Controller pattern
Select correct option:
Smalltalk
PASCAL
JAVA
C++

Question # 10 of 10 (Start time: 10:45:42 PM) Total Marks: 1
which of the following statements are same in output: 1) a = a >> 2 2) a = a / 4 3) a = a * 2
Select correct option:
(1) and (3) only
(2) and (3) only
(1) and (2) only
All procduce the same result

Question # 1 of 10 (Start time: 10:34:01 PM) Total Marks: 1
It ensures that a class only has one instance and provides a global point of access to it.

Observer Pattern
Real Pattern
None of the given

Question # 2 of 10 (Start time: 10:34:41 PM) Total Marks: 1
Anti-patterns is another concept that corresponds to common ---------- in analysis and design.
Select correct option:
mistakes ok
issues
problems
All of the given

Question # 3 of 10 (Start time: 10:35:18 PM) Total Marks: 1
Complex expressions:
Select correct option:
Make the code easy to modify
Make the code difficult to modify ok
Make the code easy to understand
Does not effect understandablity

Question # 4 of 10 (Start time: 10:35:52 PM) Total Marks: 1
The expressions with logical operators can be evaluated only from _____.
Select correct option:
Right to left
Left to right
Top to bottom
Bottom to top

Question # 5 of 10 (Start time: 10:36:40 PM) Total Marks: 1
A function should not be larger in any case and should not exceed _____ in length .
Select correct option:
Half page
One page ok
Two pages
Three Pages

Question # 6 of 10 (Start time: 10:37:20 PM) Total Marks: 1
Patterns are devices that allow programs to share knowledge about their ----------.
Select correct option:

Analysis
None of the given

Question # 7 of 10 (Start time: 10:37:56 PM) Total Marks: 1
In case of header files, construction is to avoid ------ errors. The construction should appear in the top the file header).
Select correct option:
compilation ok
run time
logical
All of the given

Question # 8 of 10 (Start time: 10:38:28 PM) Total Marks: 1
Which indent size eliminates the chance of code lines splitting?
Select correct option:
2 ok
4
3
6

Question # 9 of 10 (Start time: 10:38:59 PM) Total Marks: 1
_____ is a tool that can help us in reducing the size of individual functions.
Select correct option:
Inheritance
Modularity ok
Association
Abstraction

Question # 10 of 10 (Start time: 10:39:52 PM) Total Marks: 1
A self documenting code is a code that explains itself without the need of comments and extraneous

_____
Select correct option:
Flowcharts
UML diagrams
Process-flow state diagrams
All of the given choices ok

Select correct option:
Balancing
Mutating
Parallelizing
Factoring ok

Question # 2 of 10 (Start time: 08:38:05 PM) Total Marks: 1
In the N-Tire Architecture,the idea is to enhance scalability and ---------- by distributing both the c application using multiple server machines.
Select correct option:
Usability
Performance ok
Interoperability
None of the given

Question # 3 of 10 (Start time: 08:38:46 PM) Total Marks: 1
In case of a file servers, client requests selected records from a .....and the server transmits records to network.
Select correct option:
Local Memory
Network
Database
File ok

Question # 4 of 10 (Start time: 08:39:51 PM) Total Marks: 1
Issues like system performance, availability, scalability, and security are analyzed in _____.
Select correct option:
Logical View ok
Physical View
Code View
Concurrency View

Question # 5 of 10 (Start time: 08:40:43 PM) Total Marks: 1
A necessary supplement to transform or transaction mapping needed to create a complete architectura
Select correct option:
entity relationship diagrams
the data dictionary
processing narratives for each module
test cases for each module

Pat Client Model is one of the configurations of ……Model.
Select correct option:
Data-centered ok
Layered
Reference
Client Server

Question # 7 of 10 (Start time: 08:42:47 PM) Total Marks: 1
Defining the services of an object means:
Select correct option:
What it does? ok
What it knows?
Who knows it?
Whome it knows?

Question # 8 of 10 (Start time: 08:43:31 PM) Total Marks: 1
Aynchronous messages :
Select correct option:
are implemented as operation call
These block caller before response
occurs in multi-threaded applications ok
are shown by dotted line

Question # 9 of 10 (Start time: 08:44:28 PM) Total Marks: 1
An architectural style encompasses which of the following elements?
Select correct option:
constraints
set of components
semantic models
all of the given ok

Question # 10 of 10 (Start time: 08:45:17 PM) Total Marks: 1
The purpose of Interaction diagrams is to:
Select correct option:
Model interactions between objects
Assist in understanding how a system (a use case) actually works
Identify responsibilities/operations and assign them to classes
All of the given options ok

In case of a file servers, client requests selected records from a .....and the server transmits records to client over the network.
local memory
network
database
*file
Different messages in sequence diagrams includes:
*Both Simple and Asynchronous
Return values in Synchronous messages are represented by:
*A dotted line with label
Data-Centered Architectural Style is also called ….
*Client Server model
Issues like system performance, availability, scalability, and security are analyzed in _____.
*Physical View
The criteria used to assess the quality of an architectural design should be based on system
*data and control
Physical view in Krutchen's 4+1 architectural view model captures _____.
*Mapping(s) of the software onto the hardware and reflects its distributed aspect
Client server is a _____ system model.
*Distributed
An architectural style encompasses which of the following elements?
Missed question
Dynamic process model shows the process ……of the system.
*Structure

Tags solved quiz Solved Quizzes CS504

Facebook Twitter

You might like



CS301 Solved Quizzes File 1



CS504 Solved Quizzes File 3



CS504 Solved Quizzes

0 Comments



Add a comment...

Facebook Comments plugin

‹ Previous Post

Copyright © 2018-2021 by - VU Vicky

Software Engineering – 1

(CS504)

Lecture Notes

Delivered by

Dr. Fakhar Lodhi

© Copyright Virtual University of Pakistan

TABLE OF CONTENTS

Lecture 01: Introduction to Software Engineering……...…….. 1

Lecture 02: Introduction to Software Development …… 11

Lecture 03: Requirement Engineering-1 …… 16

Lecture 04: Requirement Engineering-2 …… 20

Lecture 05: Relation of Several components of Software Requirements …….. 28

Lecture 06: Use Case Diagram for a Library System ……... 33

Lecture 07: Source and Sink Analysis …… 40

Lecture 08: State Transition Diagrams ……... 44

Lecture 09: Typical Processes …… 53

Lecture 10: Prototyping and GUI Design …… 62

Lecture 11: Software Design …… 69

Lecture 12: Coupling and Cohesion …….. 72

Lecture 13: Object Oriented Analysis and Design ……... 83

Lecture 14: Object Oriented Analysis and Design-2 …… 89

Lecture 15: UML Object Model Notations …… 92

Lecture 16: Derivation of Object Model-Coad Methodology ……. 93

Lecture 17: Derivation of Object Model-Coad Methodology -2 ……. 95

Lecture 18: CASE STUDY: Connie's Convenience Store ……. 97

Lecture 19: Identify Structure …… 100

Lecture 20: Interaction Diagrams …….. 106

Lecture 21: Sequence Diagrams (Message Types) …….. 108

Lecture 22: Software and System Architecture …….. 115

© Copyright Virtual University of Pakistan

Lecture 23: Architectural Views …… 122

Lecture 24: Architectural Models-I ……... 126

Lecture 25: Architectural Models-II …….. 130

Lecture 26: Introduction to Design Patterns …… 137

Lecture 27: Observer Pattern ……. 140

Lecture 28: Good Programming Practices and Guidelines…… 146

Lecture 29: File Handling Tips for C++ and Java ……... 155

Lecture 30: Layouts and Comments in Java and C++…… 162

Lecture 31: Coding Style Guidelines Continued... ……..167

Lecture 32: Clarity Trough Modularity ……... 170

Lecture 33: Common Coding Mistakes …….. 176

Lecture 34: Portability …… 179

Lecture 35: Exception Handling …… 184

Lecture 36: Software Verification and Validation …….. 192

Lecture 37: Testing vs. Development ……... 195

Lecture 38: Equivalence Classes or Equivalence Partitioning …….. 199

Lecture 39: White Box Testing …….. 202

Lecture 40: Unit Testing ……. 207

Lecture 41: Inspections vs. Testing …….. 210

Lecture 42: Debugging ……... 213

Lecture 43: Bug Classes …….. 216

Lecture 44: The Holistic Approach ……... 224

Lecture 45: Summary ……. 227

© Copyright Virtual University of Pakistan

Lecture No. 1

Introduction to Software Engineering

An Introduction to Software Construction Techniques for Industrial Strength Software

1.1 Introduction

Software engineering is an interesting subject. In order to understand this subject we will need to look at a number of examples and case studies. And we will need to see how we can develop good software and how it could be improved in different scenarios? Before we move on to software engineering we need to understand what software actually is.

➢ What is Software?

When we write a program for computer we named it as software. But software is not just a program; many things other than the program are also included in software.

Some of the constituted items of software are described below.

- ***Program*:** The program or code itself is definitely included in the software.
- ***Data*:** The data on which the program operates is also considered as part of the software.
- ***Documentation*:** Another very important thing that most of us forget is documentation. All the documents related to the software are also considered as part of the software.

So the software is not just the code written in Cobol, Java, Fortran or C++. It also includes the data and all the documentation related to the program.

➢ Why is it important?

Undoubtedly software is playing a vital role in all the field of life these days. We can see many software applications being operated around us in our daily routine.

Some of the major areas in which software has played an important role are identified as under.

- *Business decision-making*: Software systems have played a major role in businesses where you have to analyze your data and on the basis of that analysis you have to make business decisions. This process of data analysis and decision-making has become very accurate and easy by the use of software.
- *Modern scientific investigation and engineering problem solving*: Scientific investigations and engineering problem solving require an intensive amount of

© Copyright Virtual University of Pakistan

calculations and data analysis. The accuracy of these analyses is also very important in scientific applications. This process has become very easy and accurate by the use of software. For example software systems are becoming more involved in bioinformatics and the process of DNA decoding is only possible by the use of software systems. Similarly many astronomical observations are being recorded and analyzed by the software systems these days.

- *Games*: We see many computer games these days that interests people of all ages. All these games are drive through software systems.
- *Embedded systems*: We see many kinds of gadgets being employed in our daily used things, like small microcontrollers used in our cars, televisions, microwave ovens etc. All these systems are controlled through the software.

Similarly in many other fields like education, office automation, Internet applications etc, software is being used. Due to its central importance and massive use in many fields it is contributing a lot in terms of economic activity started by the software products. Billions and trillions of dollars are being invested in this field throughout the world every year.

➢ Engineering

Before moving on to software engineering lets first discuss something about engineering itself. If you survey some of the dictionaries then you will find the following definition of engineering.

"The process of productive use of scientific knowledge is called engineering."

1.2 Difference between Computer Science and Software Engineering

The science concerned with putting scientific knowledge to practical use.
Webster's Dictionary

There are many engineering fields like electrical, mechanical and civil engineering. All these branches of engineering are based on physics. Physics itself is not engineering but the use of physics in making buildings, electronic devices and machines is engineering. When we use physics in constructing buildings then it is called civil engineering. When we use physics in making machines like engines or cars then it is called mechanical engineering. And when we apply the knowledge of physics in developing electronic devices then the process is called electrical engineering. The relation of computer science with software engineering is similar as the relation of physics with the electrical, mechanical or civil engineering or for that matter the relation of any basic science with any engineering field. So in this context we can define software engineering as:
"This is the process of utilizing our knowledge of computer science in effective production of software systems."

➢ Difference between Software and Other Systems

Now lets talk something about how a software system is different from any other systems. For example, how software is different from a car, a TV or the similar systems or what is the difference between software engineering and other engineering like

© Copyright Virtual University of Pakistan

mechanical or electrical engineering. Lets look at some of the non-software systems like TV, Car or an Electric Bulb. The car may be malfunctioned due to some problem in engine while driving. Similarly an electric bulb may be fused while glowing and a TV could be dysfunctional while working.

So the major thing that distinguishes a software system from other systems is that;

"Software does not wear out!"

What does that mean?
As we have seen in above example that our non-software systems could be malfunctioned or crash while working. That mean they are affected by the phenomenon of wear and tear. They have a particular life and after that they could have some problem and may not behave and perform as expected. But this is not the case with software. Software systems does not affect by the phenomenon of wear and tear. If a software has any defect then that defect will be there from the very first day and that defect normally called bug. That means if a software is not working then it should not work from the very first day. But this could not be the case that at a particular point in time a software is functioning well and after some time the same software is not performing the same task as required. So software does not have the element of wear and tear. Lets elaborate this point further. We have just talked about software defects which we call bugs. If a part of a car became wear out you just need to get a new one from market and replace the damages one with the new one. And the car will start working properly as it was working previously. Similarly if an electric bulb got fused then you just need to get a new one and put into the socket in place of the fused one and your room will again be illuminated. But the case of software is somewhat different. If a software has a bug then the same process of replacing faulty part with the new one may not work. You cannot remove the bug by just replacing the faulty part of software with the new one. Or it will not be as simple that, you go to the concerned company, get a new CD of that software and it will start working properly. If the software has a bug and that bug was present in the older CD then that will remain in the new one. This is a fundamental difference between software and other systems.

1.3 Source of Inherent Complexity of Software

Here the subject is again the same that how software systems are different from other systems. Have you ever noticed that how many different models of a car do a car manufacturing company release in a year? And how many major changes are made in new models and what is the frequency of these changes. If you think a little bit on it then you will realize that once the system is finalized then the changes in new models are of very minor nature. A drastic change is very unlikely in these kinds of systems. So the frequency of changes in these systems is very low and of minor nature. Like body shape could be changed a little, a new gadget could be added and the like but it is very unlikely that a fundamental change in engine is made. On the other hand if you observe the activities of a software manufacturing company, you will realize that these companies make changes of fundamental nature in their software systems. They constantly change their systems whether in the form of enhancements, in the form of interface change or

© Copyright Virtual University of Pakistan

they are making a new system altogether. In other words they are making changes in their systems in many different dimensions. But in non-software systems these kind of changes are not that much frequent. One of the major reasons of increased bugs in software systems is this high frequency of change. You can well imagine that if a car manufacturing company manufacture cars in the similar way then how long these cars will remain useful, how much effort they have to put to design these cars, how much time they will require to mature the design, and how much time they would be needing to start production of such cars. If they try to cut-short that time, meaning that if they try to release cars after every six-months or a year without proper testing and that release has a fundamental change then that kind of cars will also have lots of bugs and will not be road-worthy.

Therefore one of the major reasons of complexity in software is due to its basic nature that the software passes through a constant process of evolution. *The name of the game is change and evolution all the times in all the dimensions.* This change has the direct impact on software in the form of defects. Therefore software engineers also have to deals with the challenge of managing this process of change and evolution.

1.4 Software Crisis

What is Software Crisis?

Computer systems were very new and primitive in early fifties and the use of software was also very limited at that time. It was limited to some scientific applications or used to process the data of census. In 1960s a great amount of rapid improvement was made in hardware. New hardware and new computer systems were made available. These computer systems were far more powerful than the computers of early fifties. It is all relative, the computers of 1960s are primitive as compare to the computers we have these days but were far more powerful than the computers of early fifties. More powerful hardware resulted into the development of more powerful and complex software. Those very complex software was very difficult to write. So the tools and techniques that were used for less complex software became inapplicable for the more complex software. Lets try to understand this with the help of an example.

Let's imagine a person who use to live in a village and who have constructed a hut for him to live. Definitely he should have face some problems in the beginning but was managed to build a hurt for him. Now if you ask him to construct another hut, he may be able to construct one more easily and in a better way. This new hut may be better than the first one and he may construct it in a relatively less time. But if you ask him to construct concrete and iron houses then he may not be able to handle it. Since he made a hut and he know how to make a place to live so you may expect from him to build concrete and iron buildings. If this is the case then you should all agree that the building constructed by that person will not have a stable structure or he may not even be able to build one.

In early 60s software had suffered from the similar kind of problem to which we call *Software Crisis*. Techniques that were used to develop small software were not applicable for large software systems. This thing resulted in the following consequences.

© Copyright Virtual University of Pakistan

- In most of the cases that software which was tried to be build using those old tools and techniques were not complete.
- Most of the times it was delivered too late.
- Most of the projects were over-budgeted.
- And in most of the case systems build using these techniques were not reliable – meaning that they were not be able to do what they were expected to do.

As a result of these problems a conference were held in 1960 in which the term software crisis was introduced. And the major issue discussed was that the development of software is in crisis and we have not been able to handle its complexities. And the term of Software Engineering was also coined in the same conference. People have said that, we should use engineering principles in developing software in the same way as we use these principles in developing cars, buildings, electronic devices etc. Software engineering is the result of software crisis when people realized that it is not possible to construct complex software using the techniques applicable in 1960s. An important result of this thing was that people had realized that just coding is not enough.

More Complex Software Applications

This conception is also very common these days. People think that if one knows how to code then that's sufficient. But just writing code is not the whole story. People have realized this fact way back in 1960s that only coding is not sufficient to develop software systems, we also need to apply engineering principles.

Software Engineering as defined by IEEE:

Let's look at some of the definitions of software engineering.

Software Engineering as defined by IEEE (International institute of Electric and Electronic Engineering). IEEE is an authentic institution regarding the computer related issues.

"The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software."

Before explaining this definition lets first look at another definition of Software Engineering given by Ian Somerville.

"All aspects of software production' Software engineering is not just concerned with the technical processes of software development but also with activities such as software project management and with the development of tools, methods and theories to support software production".

These definitions make it clear that Software Engineering is not just about writing code.

© Copyright Virtual University of Pakistan

1.5 Software Engineering

Software Engineering is the set of processes and tools to develop software. *Software Engineering is the combination of all the tools, techniques, and processes that used in software production.* Therefore Software Engineering encompasses all those things that are used in software production like:

- Programming Language
- Programming Language Design
- Software Design Techniques
- Tools
- Testing
- Maintenance
- Development etc.

So all those thing that are related to software are also related to software engineering.

Some of you might have thought that how programming language design could be related to software engineering. If you look more closely at the software engineering definitions described above then you will definitely see that software engineering is related to all those things that are helpful in software development. So is the case with programming language design. Programming language design is one of the major successes in last fifty years. The design of Ada language was considered as the considerable effort in software engineering.

These days object-oriented programming is widely being used. If programming languages will not support object-orientation then it will be very difficult to implement object-oriented design using object-oriented principles. All these efforts made the basis of software engineering.

Well-Engineered Software

Let's talk something about what is well-engineered software. Well-engineered software is one that has the following characteristics.

- It is reliable
- It has good user-interface
- It has acceptable performance
- It is of good quality
- It is cost-effective

Every company can build software with unlimited resources but well-engineered software is one that conforms to all characteristics listed above.

Software has very close relationship with economics. Whenever we talk about engineering systems we always first analyze whether this is economically feasible or not. Therefore you have to engineer all the activities of software development while keeping its economical feasibility intact.

© Copyright Virtual University of Pakistan

The major challenges for a software engineer is that he has to build software within limited time and budget in a cost-effective way and with good quality

Therefore well-engineered software has the following characteristics.

- Provides the required functionality
- Maintainable
- Reliable
- Efficient
- User-friendly
- Cost-effective

But most of the times software engineers ends up in conflict among all these goals. It is also a big challenge for a software engineer to resolve all these conflicts.

The Balancing Act!

Software Engineering is actually the balancing act. You have to balance many things like cost, user friendliness, Efficiency, Reliability etc. You have to analyze which one is the more important feature for your software is it reliability, efficiency, user friendliness or something else. There is always a trade-off among all these requirements of software. It may be the case that if you try to make it more user-friendly then the efficiency may suffer. And if you try to make it more cost-effective then reliability may suffer. Therefore there is always a trade-off between these characteristics of software.

These requirements may be conflicting. For example, there may be tension among the following:

- Cost vs. Efficiency
- Cost vs. Reliability
- Efficiency vs. User-interface

A Software engineer is required to analyze these conflicting entities and tries to strike a balance.

Challenge is to balance these requirements.

Software Engineers always confront with the challenge to make a good balance of all these tings depending on the requirements of the particular software system at hand. He should analyze how much weight should all these things get such that it will have acceptable quality, acceptable performance and will have acceptable user-interface.

In some software the efficiency is more important and desirable. For example if we talk about a cruise missile or a nuclear reactor controller that are droved by the software systems then performance and reliability is far more important than the cost-effectiveness and user-friendliness. In these cases if your software does not react within a certain amount of time then it may result in the disaster like Chernobyl accident.

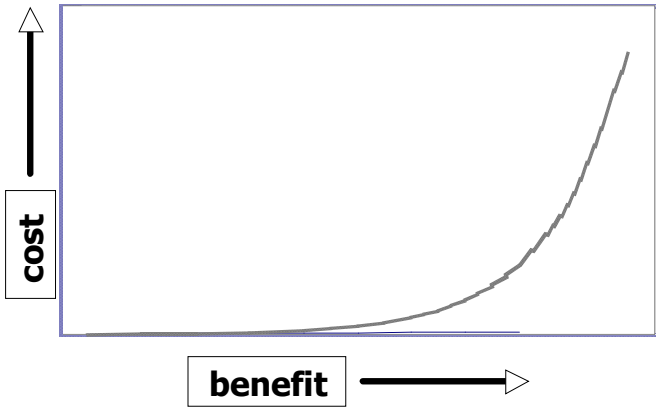
© Copyright Virtual University of Pakistan

Therefore software development is a process of balancing among different characteristics of software described in the previous section. And it is an art to come up with such a good balance and that art can be learned from experience.

Law of diminishing returns

In order to understand this concept lets take a look at an example. Most of you have noticed that if you dissolve sugar in a glass of water then the sweetness of water will increase gradually. But at a certain level of saturation no more sugar will dissolved into water. Therefore at that point of saturation the sweetness of water will not increase even if you add more sugar into it.

The law of diminishing act describes the same phenomenon. Similar is the case with software engineering. Whenever you perform any task like improving the efficiency of the system, try to improve its quality or user friendliness then all these things involve an element of cost. If the quality of your system is not acceptable then with the investment of little money it could be improved to a higher degree. But after reaching at a certain level of quality the return on investment on the system's quality will become reduced. Meaning that the return on investment on quality of software will be less than the effort or money we invest. Therefore, in most of the cases, after reaching at a reasonable level of quality we do not try to improve the quality of software any further. This phenomenon is shown in the figure below.



Software Background

Caper Jones a renounced practitioner and researcher in the filed of Software Engineering, had made immense research in software team productivity, software quality, software cost factors and other fields relate to software engineering. He made a company named Software Productivity Research in which they analyzed many projects and published the results in the form of books. Let's look at the summary of these results.

He divided software related activities into about twenty-five different categories
They have analyzed around 10000 software projects to come up with

© Copyright Virtual University of Pakistan

such a categorization. But here to cut down the discussion we will only describe nine of them that are listed below.

- Project Management
- Requirement Engineering
- Design
- Coding
- Testing
- Software Quality Assurance
- Software Configuration Management
- Software Integration and
- Rest of the activities

One thing to note here is that you cannot say that anyone of these activities is dominant among others in terms of effort putted into it. Here the point that we want to emphasize is that, though coding is very important but it is not more than 13-14% of the whole effort of software development.

Fred Brook is a renowned software engineer; he wrote a great book related to software engineering named "A Mythical Man Month". He combined all his articles in this book. Here we will discuss one of his articles named "No Silver Bullet" which he included in the book.

An excerpt from "No Silver Bullet" – Fred Brooks

> *Of all the monsters that fill the nightmares of our folklore, none terrify more than werewolves, because they transform unexpectedly from the familiar into horrors. For these we seek bullets of silver that can magically lay them to rest. The familiar software project has something of this character (at least as seen by the non-technical manager), usually innocent and straight forward, but capable of becoming a monster of missed schedules, blown budgets, and flawed projects. So we hear desperate cries for a silver bullet, something to make software costs drop as rapidly as computer hardware costs do. Skepticism is not pessimism, however. Although we see no startling breakthroughs, and indeed, such to be inconsistent with the nature of the software, many encouraging innovations are under way. A disciplined, consistent effort to develop, propagate and exploit them should indeed yield an order of magnitude improvement. There is no royal road, but there is a road. The first step towards the management of disease was replacement of demon theories and humors theories by the germ theory. The very first step, the beginning of hope, in itself dashed all hopes of magical solutions. It told workers that progress would be made stepwise, at great effort, and that a persistent, unremitting care would have to be paid to a discipline of cleanliness. So it is with software engineering today.*

So, according to Fred Brook, in the eye of an unsophisticated manager software is like a giant. Sometimes it reveals as an unscheduled delay and sometimes it shows up in the form of cost overrun. To kill this giant the managers look for magical solutions. But

© Copyright Virtual University of Pakistan

unfortunately magic is not a reality. We do not have any magic to defeat this giant. There is only one solution and that is to follow a disciplined approach to build software. We can defeat the giant named software by using disciplined and engineered approach towards software development.

Therefore, *Software Engineering is nothing but a disciplined and systematic approach to software development*.

1.6 Summary

Today we have discussed the following things related to software engineering.

- What is software engineering?
- Why is it important?
- What is software crisis?
- How software engineering derived from software crisis.
- What is the importance of engineering principles in developing software?
- What is balancing act and how apply in software engineering?
- What is law of diminishing returns?
- And what are the major activities involved in the development of software.

© Copyright Virtual University of Pakistan

Lecture No. 02

Introduction to Software Development

2.1 Software Development

We have seen in our previous discussion that software engineering is nothing but a disciplined approach to develop software. Now we will look at some of the activities involved in the course of software development. The activities involved in software development can broadly be divided into two major categories first is construction and second is management. The construction activities are those that are directly related to the construction or development of the software. While the management activities are those that complement the process of construction in order to perform construction activities smoothly and effectively. A greater detail of the activities involved in the construction and management categories is presented below.

Construction

The construction activities are those that directly related to the development of software, e.g. gathering the requirements of the software, develop design, implement and test the software etc. Some of the major construction activities are listed below.

- Requirement Gathering
- Design Development
- Coding
- Testing

Management

Management activities are kind of umbrella activities that are used to smoothly and successfully perform the construction activities e.g. project planning, software quality assurance etc. Some of the major management activities are listed below.

- Project Planning and Management
- Configuration Management
- Software Quality Assurance
- Installation and Training

As we have said earlier that management activities are kind of umbrella activities that surround the construction activities so that the construction process may proceed smoothly. This fact is empathized in the figure 1. The figure shows that construction is surrounded by management activities. That is, all construction activities are governed by certain processes and rules. These processes and rules are related to the management of the construction activities and not the construction itself.

© Copyright Virtual University of Pakistan

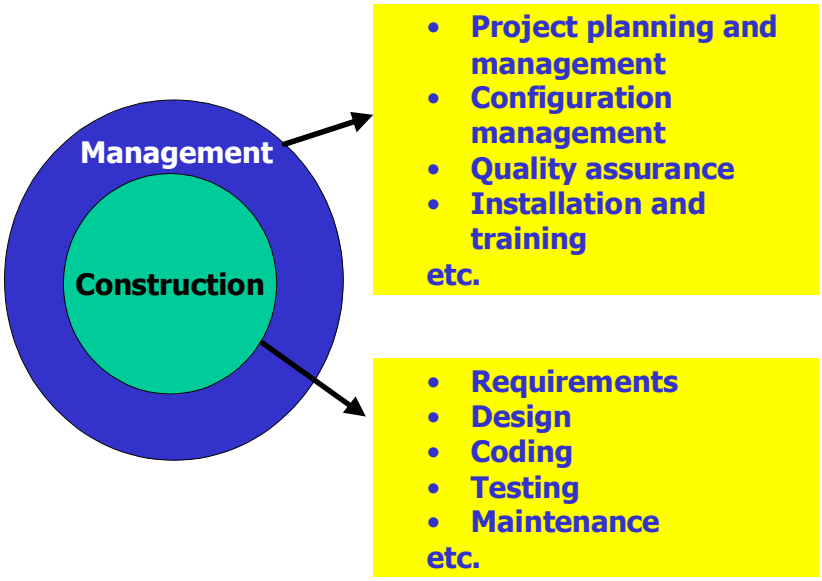


Figure1: Development activities

2.2 A Software Engineering Framework

Any Engineering approach must be founded on organizational commitment to quality. That means the software development organization must have special focus on quality while performing the software engineering activities. Based on this commitment to quality by the organization, a software engineering framework is proposed that is shown in figure 2. The major components of this framework are described below.

Quality Focus: As we have said earlier, the given framework is based on the organizational commitment to quality. The quality focus demands that processes be defined for rational and timely development of software. And quality should be emphasized while executing these processes.

Processes: The processes are set of key process areas (KPAs) for effectively manage and deliver quality software in a cost effective manner. The processes define the tasks to be performed and the order in which they are to be performed. Every task has some deliverables and every deliverable should be delivered at a particular milestone.

Methods: Methods provide the technical "how-to's" to carryout these tasks. There could be more than one technique to perform a task and different techniques could be used in different situations.

Tools: Tools provide automated or semi-automated support for software processes, methods, and quality control.

© Copyright Virtual University of Pakistan

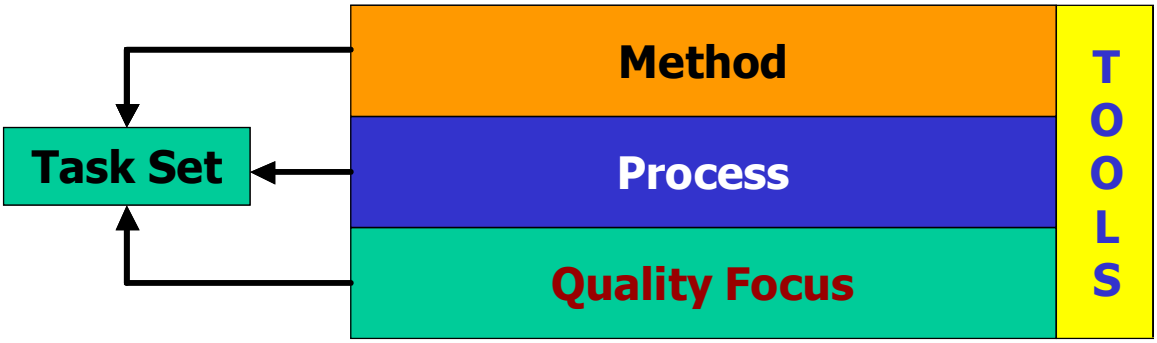


Figure 2: A Software Engineering Framework

2.3 Software Development Loop

Lets now look at software engineering activities from a different perspective. Software development activities could be performed in a cyclic and that cycle is called software development loop which is shown in figure 3. The major stages of software development loop are described below.

Problem Definition: In this stage we determine what is the problem against which we are going to develop software. Here we try to completely comprehend the issues and requirements of the software system to build.

Technical Development: In this stage we try to find the solution of the problem on technical grounds and base our actual implementation on it. This is the stage where a new system is actually developed that solves the problem defined in the first stage.

Solution Integration: If there are already developed system(s) available with which our new system has to interact then those systems should also be the part of our new system. All those existing system(s) integrate with our new system at this stage.

Status Quo: After going through the previous three stages successfully, when we actually deployed the new system at the user site then that situation is called status quo. But once we get new requirements then we need to change the status quo.

After getting new requirements we perform all the steps in the software development loop again. The software developed through this process has the property that this could be evolved and integrated easily with the existing systems.

© Copyright Virtual University of Pakistan

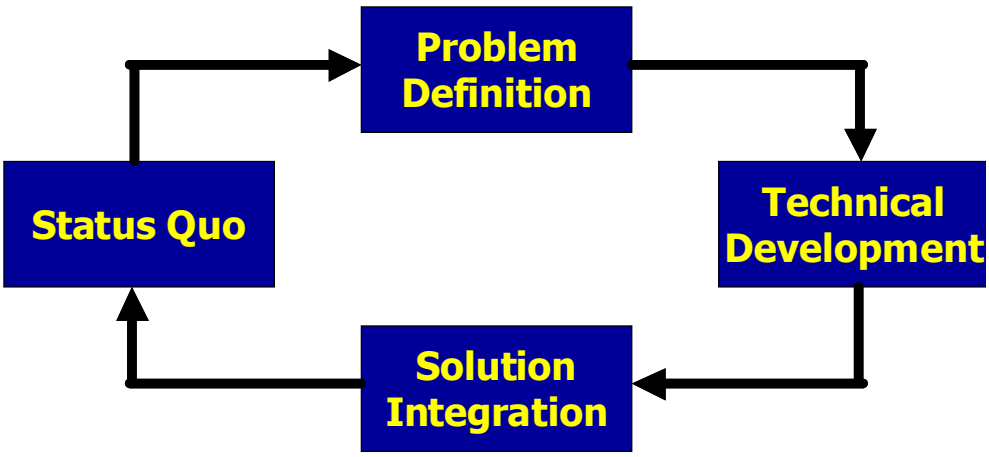


Figure 3: Software Development Loop

Software Construction

Here once again look at the construction activities of the software from a different perspective. This section provides with a sequence of questions that have to answer in different stages of software development.

1. What is the problem to be solved?
2. What are the characteristics of the entity that is used to solve the problem?
3. How will the entity be realized?
4. How will the entity be constructed?
5. What approach will be used to uncover errors that were made in the design and construction of the entity?
6. How will the entity be supported over the long term, when users of the entity request corrections, adaptations, and enhancements?

2.4 Software Engineering Phases

There are four basic phases of software development that are shown in Figure 4.

Vision: Here we determine why are we doing this thing and what are our business objectives that we want to achieve.

Definition: Here we actually realize or automate the vision developed in first phase. Here we determine what are the activities and things involved.

Development: Here we determine, what should be the design of the system, how will it be implemented and how to test it.

Maintenance: This is very important phase of software development. Here we control the change in system, whether that change is in the form of enhancements or defect removel.

© Copyright Virtual University of Pakistan

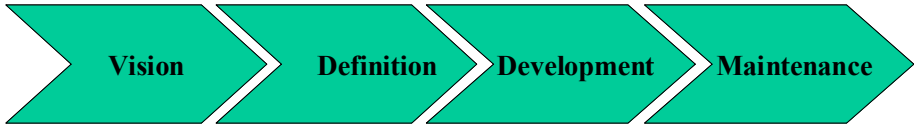


Figure 4: Software Engineering Phases

Maintenance

Correction, adaptation, enhancement
For most large, long lifetime software systems, maintenance cost normally exceeds development cost by factors ranging from 2 to 3.

Boehm (1975) quotes a pathological case where the development cost of an avionics system was $30 per line of code but the maintenance cost was $4000 per instruction

2.5 Summary

- Software development is a multi-activity process. It is not simply coding.
- Software construction and management
- Software Engineering Framework
- Software development loop
- Software engineering phases
- Importance of Maintenance

© Copyright Virtual University of Pakistan

Lecture No. 3

Requirement Engineering

3.1 Requirement Engineering

We recall from our previous discussion that software development is not simply coding – it is a multi-activity process. The process of software construction encompasses and includes answers to the following questions:

- What is the problem to be solved?
- What are the characteristics of the entity that is used to solve the problem?
- How will the entity be realized?
- How will the entity be constructed?
- What approach will be used to uncover errors that were made in the design and construction of the entity?
- How will the entity be supported over the long term when users of the entity request corrections, adaptations, and enhancements?

These questions force us to look at the software development process from different angles and require different tools and techniques to be adopted at different stages and phases of the software development life cycle. Hence we can divide the whole process in 4 distinct phases namely vision, definition, development, and maintenance. Each one of these stages has a different focus of activity. During the vision phases, the focus is on why do we want to have this system; during definition phase the focus shifts from why to what needs to be built to fulfill the previously outlined vision; during development the definition is realized into design and implementation of the system; and finally during maintenance all the changes and enhancements to keep the system up and running and adapt to the new environment and needs are carried out.

Requirement engineering mainly deals with the definition phase of the system. Requirement engineering is the name of the process when the system services and constraints are established. It is the starting point of the development process with the focus of activity on what and not how.

Software Requirements Definitions

Before talking about the requirement process in general and discussing different tools and techniques used for developing a good set of requirements, let us first look at a few definitions of software requirements.

Jones defines software requirements as a statement of needs by a user that triggers the development of a program or system.

© Copyright Virtual University of Pakistan

Alan Davis defines software requirements as a user need or necessary feature, function, or attribute of a system that can be sensed from a position external to that system.

According to Ian Summerville, requirements are a specification of what should be implemented. They are descriptions of how the system should behave, or of a system property or attribute. They may be a constraint on the development process of the system.

IEEE defines software requirements as:

1. **A condition or capability needed by user to solve a problem or achieve an objective.**
2. A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed document.
3. A documented representation of a condition or capability as in 1 or 2.

As can be seen, these definitions slightly differ from one another but essentially say the same thing: a software requirement is a document that describes all the services provided by the system along with the constraints under which it must operate.

3.2 Importance of Requirements

Many of the problems encountered in SW development are attributed to shortcoming in requirement gathering and documentation process. We cannot imagine start building a house without being fully satisfied after reviewing all the requirements and developing all kinds of maps and layouts but when it comes to software we really do not worry too much about paying attentions to this important phase. This problem has been studied in great detail and has been noted that 40-60% of all defects found in software projects can be traced back to poor requirements.

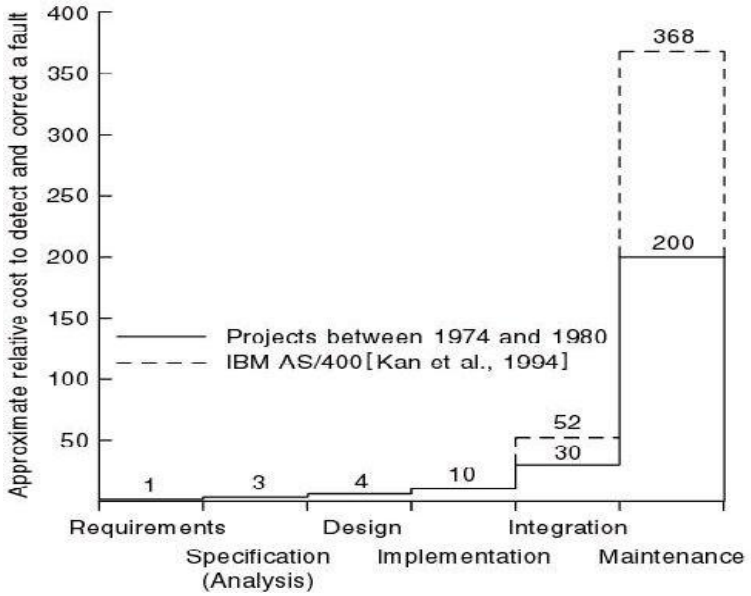
Fred Brooks in his classical book on software engineering and project management "The Mythical Man Month" emphasizes the importance of requirement engineering and writes:

> *"The hardest single part of building a software system is deciding precisely what to build. No other part of the conceptual work is as difficult as establishing the detailed technical requirements, including all the interfaces to people, to machines, and to other software systems. No other part of the work so cripples the system if done wrong. No other part is more difficult to rectify later."*

Let us try to understand this with the help of an analogy of a house. If we are at an advanced stage of building a house, adding a new room or changing the dimensions of some of the rooms is going to be very difficult and costly. On the other hand if this need is identified when the maps are being drawn, one can fix it at the cost of redrawing the map only. In the case of a software development, we experience the exact same phenomenon - if a problem is identified and fixed at a later stage in the software development process, it will cost much more than if it was fixed at and earlier stage.

© Copyright Virtual University of Pakistan

This following graph shows the relative cost of fixing problem at the various stages of software development.



Boehm (1981) has reported that correcting an error after development costs 68 times more. Other studies suggest that it can be as high as 200 times. Since cost is directly related with the success or failure of projects, it is clear from all this discussion that having sound requirements is the most critical success factor for any project.

3.3 Role of Requirements

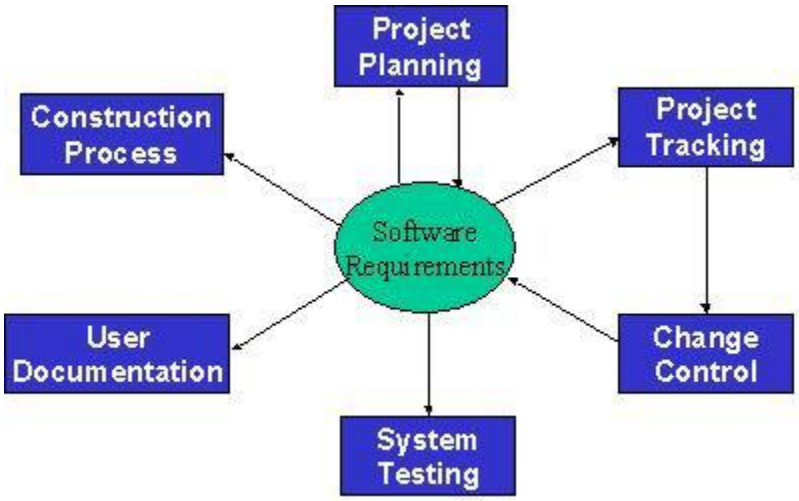
Software requirements document plays the central role in the entire software development process. To start with, it is needed in the project planning and feasibility phase. In this phase, a good understanding of the requirements is needed to determine the time and resources required to build the software. As a result of this analysis, the scope of the system may be reduced before embarking upon the software development.

Once these requirements have been finalized, the construction process starts. During this phase the software engineer starts designing and coding the software. Once again, the requirement document serves as the base reference document for these activities. It can be clearly seen that other activities such as user documentation and testing of the system would also need this document for their own deliverables.

On the other hand, the project manager would need this document to monitor and track the progress of the project and if needed, change the project scope by modifying this document through the change control process.

© Copyright Virtual University of Pakistan

The following diagram depicts this central role of the software requirement document in the entire development process.



© Copyright Virtual University of Pakistan

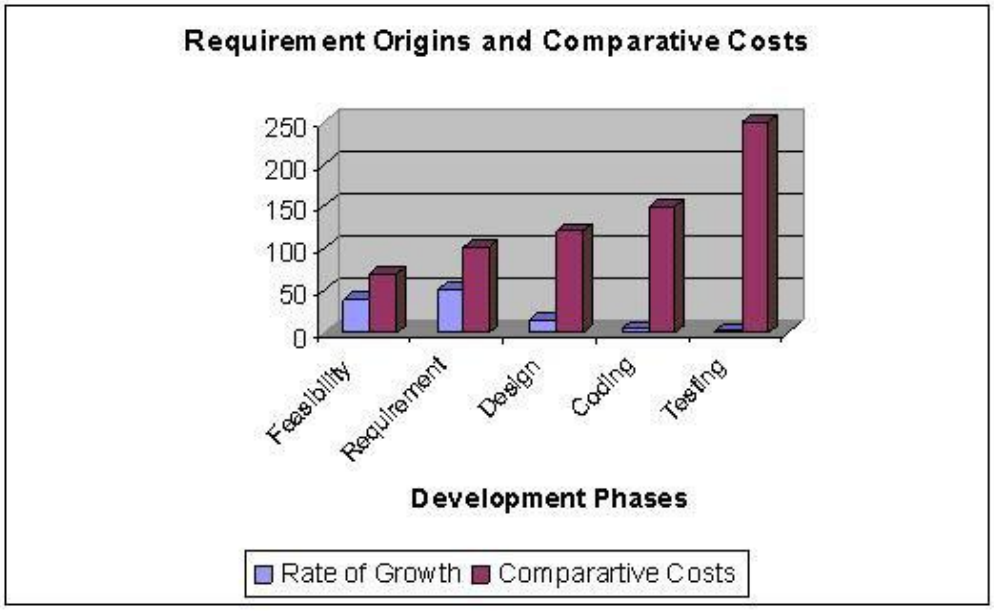
Lecture No. 4
Requirement Engineering-2

3.4 Some Risks from Inadequate Requirement Process

From the above discussion, it should be clear that the requirements play the most significant role in the software development process and the success and failure of a system depends to a large extent upon the quality of the requirement documents. Following is a list of some of the risks of adopting an inadequate requirement process:

1. Insufficient user involvement leads to unacceptable products.
   If input from different types of user is not taken, the output is bound to lack in key functional areas, resulting in an unacceptable product. Overlooking the needs of certain user classes (stake holders) leads to dissatisfaction of customers.

2. Creeping user requirements contribute to overruns and degrade product quality.
   Requirement creep is one of the most significant factors in budget and time overruns. It basically means identifying and adding new requirements to the list at some advanced stages of the software development process. The following figure shows the relative cost of adding requirements at different stages.



© Copyright Virtual University of Pakistan

3. Ambiguous requirements lead to ill-spent time and rework.

   Ambiguity means that two different readers of the same document interpret the requirement differently. Ambiguity arises from the use of natural language. Because of the imprecise nature of the language, different readers interpret the statements differently. As an example, consider the following Urdu Phrase: *"Rooko mut jane doo"*. Now, depending upon where a reader places the comma in this statement, two different readers may interpret it in totally different manner. If a comma is palced after *"Rooko"*, the sentence will become *"Rooko, mut jane doo"*, meaning *"don't let him go"*. On the other hand if the comma id placed after *"mut"*, the sentence will become *"Rooko mut, jane doo"*, meaning *"let him go"*. Ambiguous requirements therefore result in misunderstandings and mismatched expectations, resulting in a wasted time and effort and an undesirable product.

   Let us consider the following requirement statement:

   The operator identity consists of the operator name and password; the password consists of six digits. It should be displayed on the security VDU and deposited in the login file when an operator logs into the system.

   This is an example of ambiguous requirement as it is not clear what is meant by *"it"* in the second sentence and what should be displayed on the VDU. Does it refer to the operator identity as a whole, his name, or his password?

4. Gold-plating by developers and users adds unnecessary features.

   Gold-plating refers to features are not present in the original requirement document and in fact are not important for the end-user but the developer adds them anyway thinking that they would add value to the product. Since these features are outside the initial scope of the product, adding them will result in schedule and budget overruns.

© Copyright Virtual University of Pakistan

5. Minimal specifications lead to missing key requirements and hence result in an unacceptable product.

   As an example, let us look at the following requirement. The requirement was stated as: "*We need a flow control and source control engineering tool."* Based upon this requirement, system was built. It worked perfectly and had all the functionality needed for source control engineering tool and one could draw all kinds of maps and drawings. The system however could not be used because there was there was no print functionality.

   Let us now look at the following set of requirement statements for another system:

   - *The system should maintain the hourly level of reservoir from depth sensor situated in the reservoir. The values should be stored for the past six months.*
   - *AVERAGE: Average command displays the average water level for a particular sensor between two times.*

   This is another case of minimal requirements – it does not state how the system should respond if we try to calculate the average water level beyond the past six months.

6. Incompletely defined requirements make accurate project planning and tracking impossible.

Levels of Software Requirements

Software requirements are defined at various levels of detail and granularity. Requirements at different level of detail also mean to serve different purposes. We first look at these different levels and then will try to elaborate the difference between these with the help of different examples.

1. **Business Requirements:**
   These are used to state the high-level business objective of the organization or customer requesting the system or product. They are used to document main system features and functionalities without going into their nitty-gritty details. They are captured in a document describing the project vision and scope.

© Copyright Virtual University of Pakistan

2. **User Requirements:**
   User requirements add further detail to the business requirements. They are called user requirements because they are written from a user's perspective and the focus of user requirement describe tasks the user must be able to accomplish in order to fulfill the above stated business requirements. They are captured in the requirement definition document.

3. **Functional Requirements:**
   The next level of detail comes in the form of what is called functional requirements. They bring-in the system's view and define from the system's perspective the software functionality the developers must build into the product to enable users to accomplish their tasks stated in the user requirements - thereby satisfying the business requirements.

4. **Non-Functional Requirements**
   In the last section we defined a software requirement as a document that describes all the services provided by the system along with the constraints under which it must operate. That is, the requirement document should not only describe the functionality needed and provided by the system, but it must also specify the constraints under which it must operate. Constraints are restrictions that are placed on the choices available to the developer for design and construction of the software product. These kinds of requirements are called Non-Functional Requirements. These are used to describe external system interfaces, design and implementation constraints, quality and performance attributes. These also include regulations, standards, and contracts to which the product must conform.

Non-functional requirement play a significant role in the development of the system. If not captured properly, the system may not fulfill some of the basic business needs. If proper care is not taken, the system may collapse. They dictate how the system architecture and framework. As an example of non-functional requirements, we can require software to run on Sun Solaris Platform. Now it is clear that if this requirement was not captured initially and the entire set of functionality was built to run on Windows, the system would be useless for the client. It can also be easily seen that this requirement would have an impact on the basic system architecture while the functionality does not change.

While writing these requirements, it must always be kept in mind that all functional requirements must derive from user requirements, which must themselves be aligned with business requirements. It must also be remembered that during the requirement engineering process we are in the definition phase of the software development where the focus is on what and not how. Therefore, requirements must not include design or implementation details and the focus should always remain on what to build and not how to build.

© Copyright Virtual University of Pakistan

Let us now look at an example to understand the difference between these different types of requirements.

Let us assume that we have a word-processing system that does not have a spell checker. In order to be able to sell the product, it is determined that it must have a spell checker. Hence the business requirement could be stated as: *user will be able to correct spelling errors in a document efficiently.* Hence, the Spell checker will be included as a feature in the product.

In the next step we need to describe what tasks must be included to accomplish the above-mentioned business requirement. The resulting user requirement could be as follows: *finding spelling errors in the document and decide whether to replace each misspelled word with the one chosen from a list of suggested words.* It is important to note that this requirement is written from a user's perspective.

After documenting the user's perspective in the form of user requirements, the system's perspective: what is the functionality provided by the system and how will it help the user to accomplish these tasks. Viewed from this angle, the functional requirement for the same user requirement could be written as follows: *the spell checker will find and highlight misspelled words. It will then display a dialog box with suggested replacements. The user will be allowed to select from the list of suggested replacements. Upon selection it will replace the misspelled word with the selected word. It will also allow the user to make global replacements.*

Finally, a non-functional requirement of the system could require that *it must be integrated into the existing word-processor that runs on windows platform.*

Stakeholders

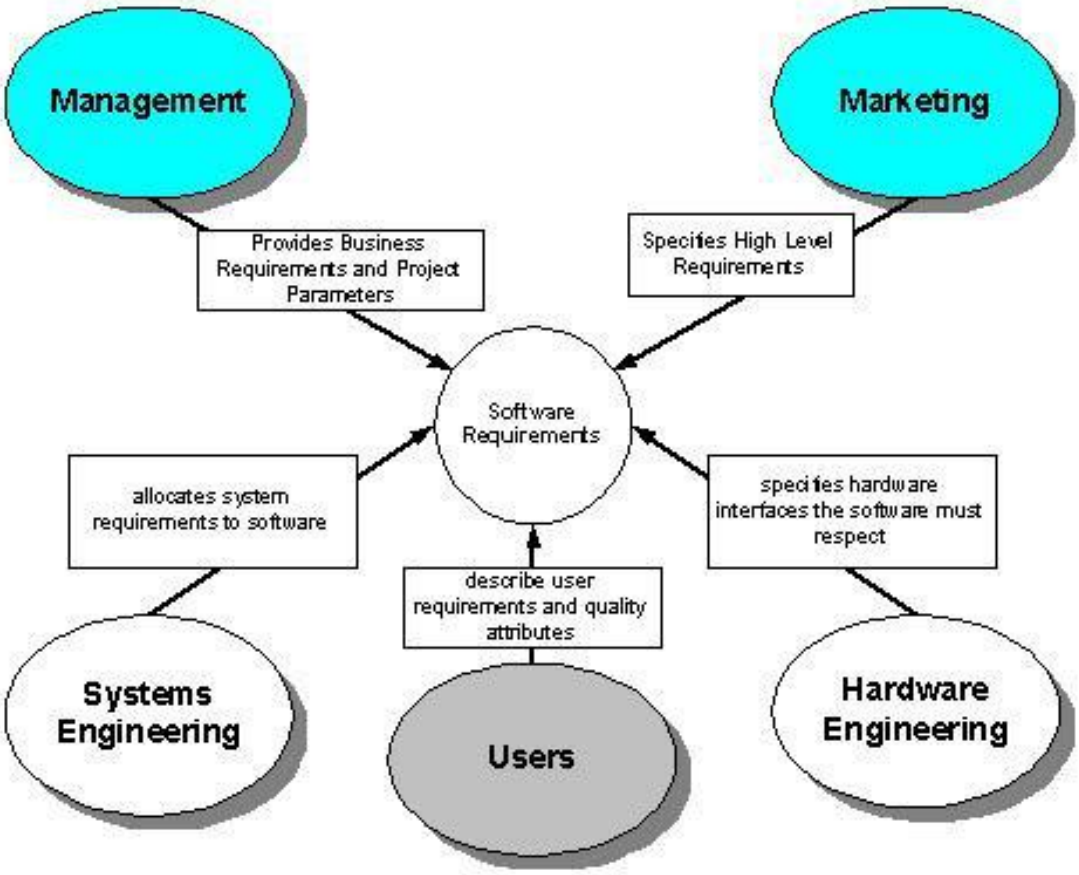
As mentioned earlier, in order to develop a good requirement document, it is imperative to involve all kinds of user in the requirement engineering process. The first step in fulfillment of this need is the identification of all the stakeholders in the system. Stakeholders are different people who would be interested in the software. It is important to recognize that management carries a lot of weight, but they usually are not the actual users of the system. We need to understand that it is the actual user who will eventually use the system and hence accept or reject the product. Therefore, ignoring the needs of any user class may result in the system failure.

A requirement engineer should be cognizant of the fact that stakeholders have a tendency to state requirements in very general and vague terms. Some times they trivialize things. Different stakeholders have different requirements – sometimes even conflicting. On top of that internal politics may influence requirements.

The role of stakeholders cannot be overemphasized. A study of over 8300 projects revealed that the top two reasons for any project failure are lack of user input and incomplete requirements.

The following diagram shows the role of different stakeholders in the setting the system requirements.

© Copyright Virtual University of Pakistan



Requirement Statement and Requirement Specification Documents

Different levels of software requirements are documented in different documents. The two main documents produced during this phase are Requirement Statement and Requirement Specification. They are also called Requirement Definition and Functional Specification and are used to document user requirements and functional requirements respectively.

Requirement Statement Characteristics

A good Requirements statement document must possess the following characteristics.

- **Complete -** Each requirement must fully describe the functionality to be delivered.

- **Correct -** Each requirement must accurately describe the functionality to be built.
- **Feasible -** It must be possible to implement each requirement within the known capabilities and limitations of the system and its environment.

© Copyright Virtual University of Pakistan

- **Necessary -**Each requirement should document something that the customer really need or something that is required for conformance to an external system requirement or standard.

- **Prioritized -** An implementation priority must be assigned to each requirement, feature or use case to indicate how essential it is to a particular product release.

- **Unambiguous -** All readers of a requirement statement should arrive at a single, consistent interpretation of it.

- **Verifiable –** User should be able to devise a small number of tests or use other verification approaches, such as inspection or demonstration, to determine whether the requirement was properly implemented.

Requirement Specification Characteristics

A good Requirements specification document should possess the following characteristics.

- **Complete -** No requirement or necessary information should be missing.

- **Consistent –** No requirement should conflict with other software or higher-level system or business requirements.

  Let us try to understand this with the help of some examples. The following set of (non-functional) requirements was stated for a particular embedded system.

  - *All programs must be written in Ada*
  - *The program must fit in the memory of the embedded micro-controller*

  These requirements conflicted with one another because the code generated by the Ada compiler was of a large footprint that could not fit into the micro-controller memory.

  Following is another set of (functional) requirements that conflicted with one another:

  - *System must monitor all temperatures in a chemical reactor.*

© Copyright Virtual University of Pakistan

- *System should only monitor and log temperatures below -20^0 C and above 400^0 C.*

In this case the two requirements clearly conflict with each other in stating what information needs to be monitored and stored.

- **Modifiable -** One must be able to revise the Software Requirement Specification when necessary and maintain a history of changes made to each requirement.

- **Traceable -** One should be able to link each requirement to its origin and to the design elements, source code, and test cases that implement and verify the correct implementation of the requirement.

Mixed level of Abstraction

It is important to recognize that all requirements in a requirement document are stated at a uniform level of abstraction. This difference in detail falsely implies the relative importance of these requirements and hence misguides all involved in the development process. The following set of requirements clearly demonstrates violation of this principle:

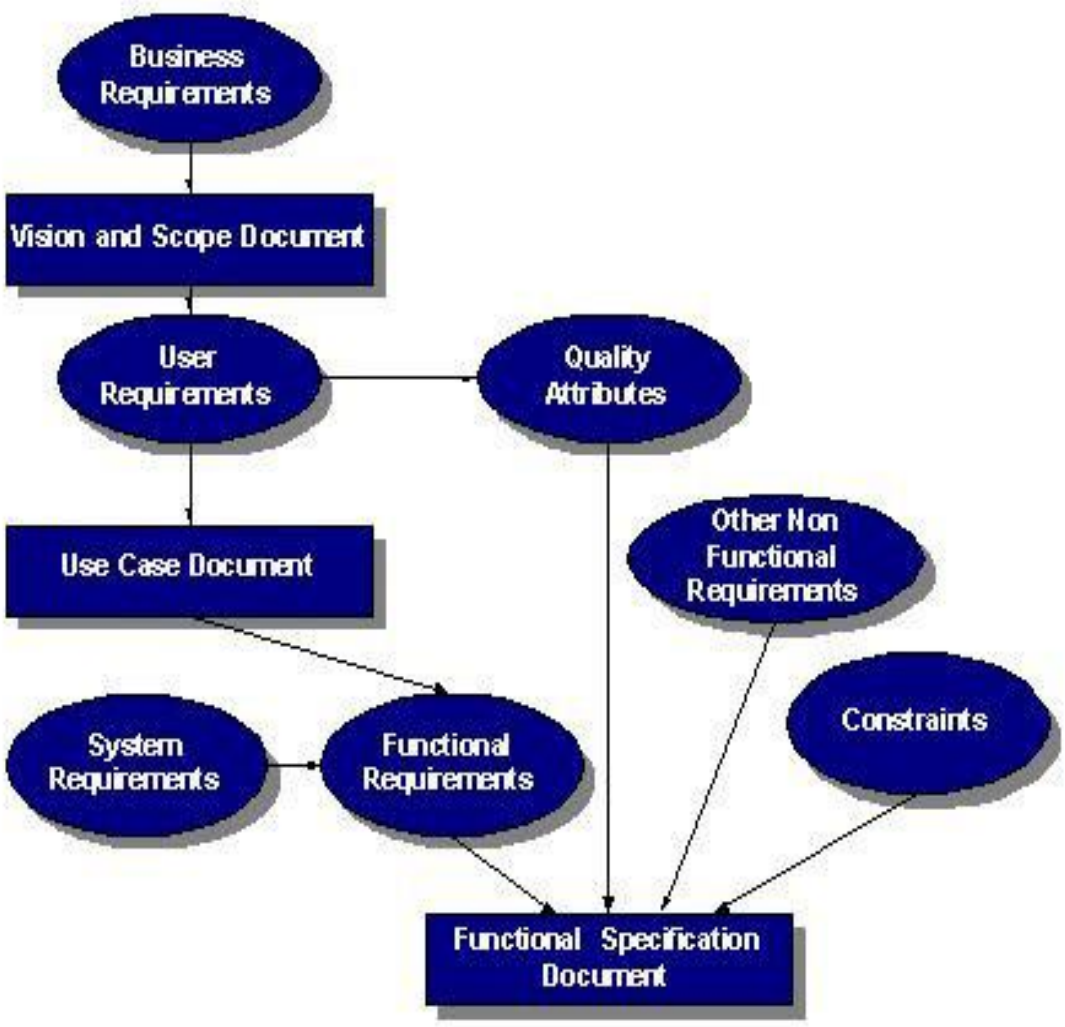
- *The purpose of the system is to track the stock in a warehouse.*
- *When a loading clerk types in the withdraw command he or she will communicate the order number, the identity of the item to be removed, and the quantity removed. The system will respond with a confirmation that the removal is allowable.*

© Copyright Virtual University of Pakistan

Lecture No. 5

Relationship of Several components of Software Requirements

The following figure depicts the relationship between different documents produced during the requirement engineering phase.



© Copyright Virtual University of Pakistan

Business Requirements

Business requirements collected from multiple sources might conflict. For example, consider a kiosk product with embedded software that will be sold to retail stores and used by the store's customers. The kiosk developer's business objectives include the following:

- leasing or selling the kiosk to the retailers
- selling consumables through the kiosk to the customer
- attracting customer to the brand
- modifying the nature of the historical developer-customer relationship

The retailer's business interest could include:

- making money from customer use of kiosk
- attracting more customers to the store
- saving money if the kiosk replaces manual operations

The developer might want to establish a high-tech and exciting new direction for customers, while the retailer wants a simple solution and the customer wants convenience and features. The tension among these three parties with their different goals, constraints, and cost factors can lead to conflicting business requirements, which must be resolved before the kiosk's software requirements are detailed.

You can also use the business requirements to set implementation priorities for use cases and their associated functional requirements. For example, a business requirement to generate maximum revenue from the kiosk would imply the early implementation of features directly associated with selling more products or services to the customer, rather than glitzy features that appeal to only a subset of customers.

The Vision Statement

The vision statement should reflect a balanced view that will satisfy the need of diverse customers. It can be somewhat idealistic but should be grounded in the realities of existing or anticipated customer markets, enterprise architectures, organizational strategic directions, and resource limitations.

© Copyright Virtual University of Pakistan

Chemical Tracking System

The chemical tracking system will allow scientists to request containers of chemicals to be supplied by chemical stockroom or by vendors. The location of every chemical container within the company, the quantity of the material remaining in it, and the complete history of each container's location and usage will be known by the system at all times. The company will save 25% on chemical costs by fully exploiting chemicals already available within the company, by disposing of fewer partially used or expired containers, and by using a standard chemical purchasing process. The chemical tracking system will also generate all reports required to comply with federal and state government regulations that require the reporting of chemical usage, storage, and disposal.

Assumptions and Dependencies

All assumptions that were made when conceiving the project have to be recorded. For example, the management sponsor for the chemical tracking system assumed that it would replace the existing chemical stockroom inventory system and that it would interface to the appropriate purchasing department applications

Scope

Project scope defines the concept and range of the proposed solution, and limitations identify certain capabilities that the product will not include. Clarifying the scope and limitations helps to establish realistic stakeholder's expectations. Sometimes customers request features that are too expansive or do not lie within the intended project scope. Propose requirements that are out of scope must be rejected, unless they are so beneficial that the scope should be enlarged to accommodate them (with accompanying changes in budget, schedule, and staff). Keep a record of these requirements and why they were rejected, as they have a way of reappearing.

Scope and Initial Release

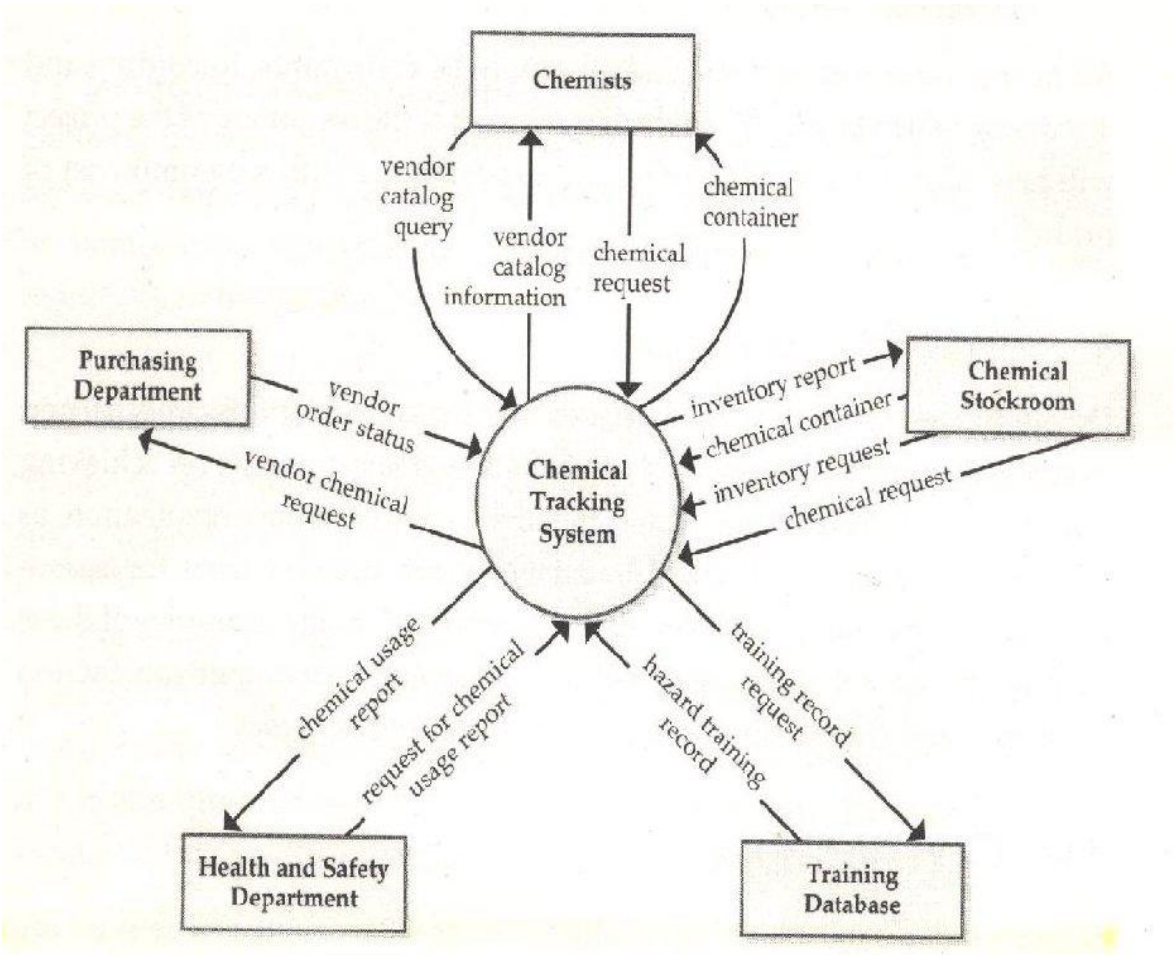
The major features that will be included in the initial release of the project should be summarized. Describe the quality characteristics that will enable the product to provide the intended benefits to its various customer communities.
Requirements need to be prioritized and a release schedule should be made.

© Copyright Virtual University of Pakistan

The Context Diagram

The scope description establishes the boundary between the system we are developing and everything else in the universe. The context diagram graphically illustrates this boundary by showing the connections between the system being developed or the problem being addressed, and the outside world. The context diagram identifies the entities outside the system that interface to it in some way (called terminators or external entities), as well as the flow of data and material between each external entity and the system. The context diagram is used as the top level abstraction in a dataflow diagram developed according to principles of structured analysis. The context diagram can be included in the vision and scope document, in the SRS, or as part of a dataflow model of the system.

Following is a context diagram of the chemical tracking system.



© Copyright Virtual University of Pakistan

Use Cases and Customer-Developer Relationship

It has been mentioned earlier on, excellent software products are the result of a well-executed design based on excellent requirements and high quality requirements result from effective communication and coordination between developers and customers. That is, good customer-developer relationship and effective communication between these two entities is a must for a successful software project. In order to build this relationship and capture the requirements properly, it is essential for the requirement engineer to learn about the business that is to be automated.

It is important to recognize that a software engineer is typically not hired to solve a computer science problem – most often than not, the problem lies in a different domain than computer science and the software engineer must understand it before it can be solved. In order to improve the communication level between the vendor and the client, the software engineer should learn the domain related terminology and use that terminology in documenting the requirements. Document should be structured and written in a way that the customer finds it easy to read and understand so that there are no ambiguities and false assumption.

One tool used to organize and structure the requirements is such a fashion is called use case modeling.

It is modeling technique developed by Ivar Jacobson to describe what a new system should do or what an existing system already does. It is now part of a standard software modeling language known as the Unified Modeling Language (UML). It captures a discussion process between the system developer and the customer. It is widely used because it is comparatively easy to understand intuitively – even without knowing the notation. Because of its intuitive nature, it can be easily discussed with the customer who may not be familiar with UML, resulting in a requirement specification on which all agree.

3.8 Use Case Model Components

A use case model has two components, use cases and actors.

In a use case model, boundaries of the system are defined by functionality that is handled by the system. Each use case specifies a complete functionality from its initiation by an actor until it has performed the requested functionality. An actor is an entity that has an interest in interacting with the system. An actor can be a human or some other device or system.

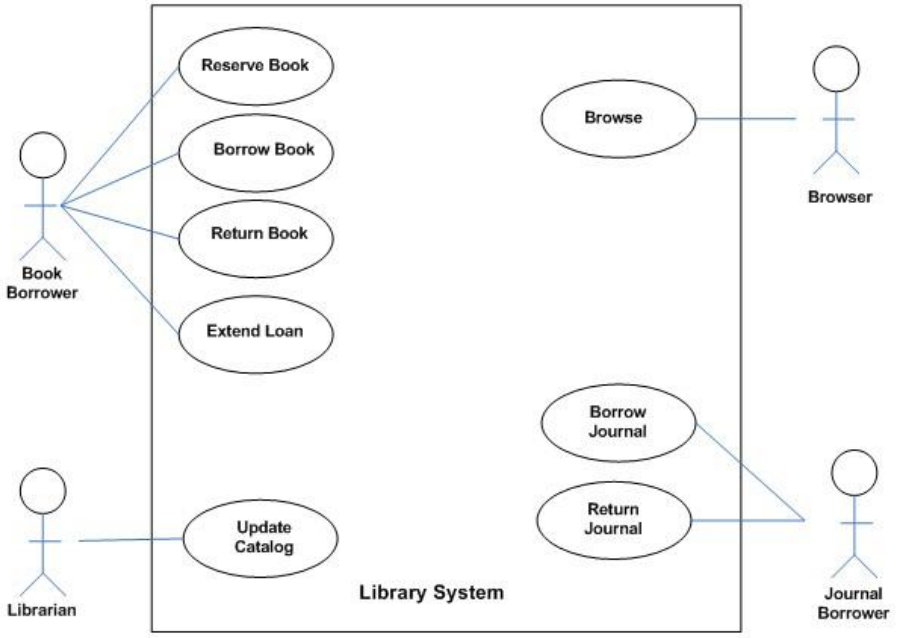
A use case model represents a use case view of the system – how the system is going to be used. In this case system is treated as a black box and it only depicts the external interface of the system. From an end-user's perspective it and describes the functional requirements of the system. To a developer, it gives a clear and consistent description of what the system should do. This model is used and elaborated throughout the development process. As an aid to the tester, it provides a basis for performing system tests to verify the system. It also provides the ability to trace functional requirements into actual classes and operations in the system and hence helps in identifying any gaps.

© Copyright Virtual University of Pakistan

Lecture No. 6

Use Diagram for a Library System

As an example, consider the following use case diagram for a library management system. In this diagram, there are four actors namely Book Borrower, Librarian, Browser, and Journal Borrower. In addition to these actors, there are 8 use cases. These use cases are represented by ovals and are enclosed within the system boundary, which is represented by a rectangle. It is important to note that every use case must always deliver some value to the actor.



With the help of this diagram, it can be clearly seen that a Book Borrower can reserve a book, borrow a book, return a book, or extend loan of a book. Similarly, functions performed by other users can also be examined easily.

Creating a Use Case Model

Creating a use case model is an iterative activity. The iteration starts with the identification of actors. In the next step, use cases for each actor are determined which define the system. After that, relationships among use cases are defined. It must be understood that these are not strictly sequential steps and it is not necessary that all actors must be identified before defining their use cases. These activities are sort of parallel and

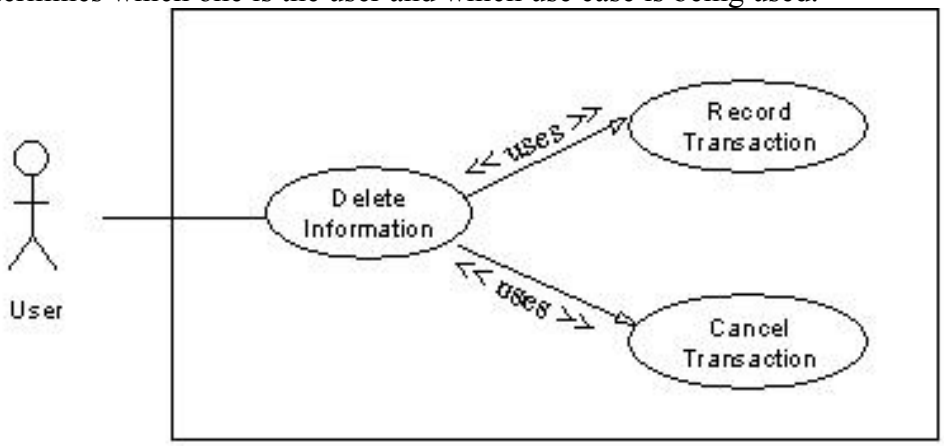
© Copyright Virtual University of Pakistan

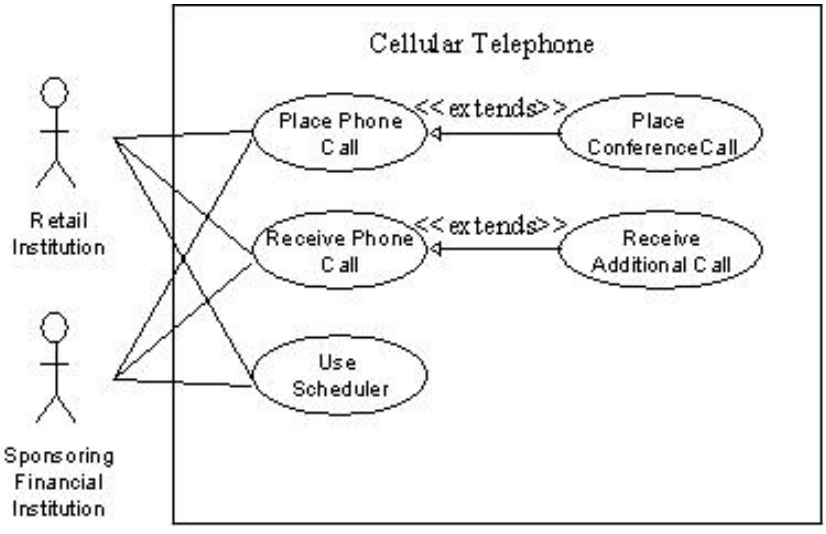
concurrent and a use case model will evolve slowly from these activities. This activity stops when no new use cases or actors are discovered. At the end, the model is validated.

3.9 Relationship among Use Cases

The UML allows us to extend and reuse already defined use cases by defining the relationship among them. Use cases can be reused and extended in two different fashions: extends and uses. In the cases of *"uses"* relationship, we define that one use case invokes the steps defined in another use case during the course of its own execution. Hence this defines a relationship that is similar to a relationship between two functions where one makes a call to the other function. The *"extends"* relationship is kind of a generalization-specialization relationship. In this case a special instance of an already existing use case is created. The new use case inherits all the properties of the existing use case, including its actors.

Let is try to understand these two concepts with the help of the following diagrams. In the case of the first diagram, the *Delete Information* use case is using two already existing use cases namely *Record Transaction* and *Cancel Transaction*. The direction of the arrow determines which one is the user and which use case is being used.



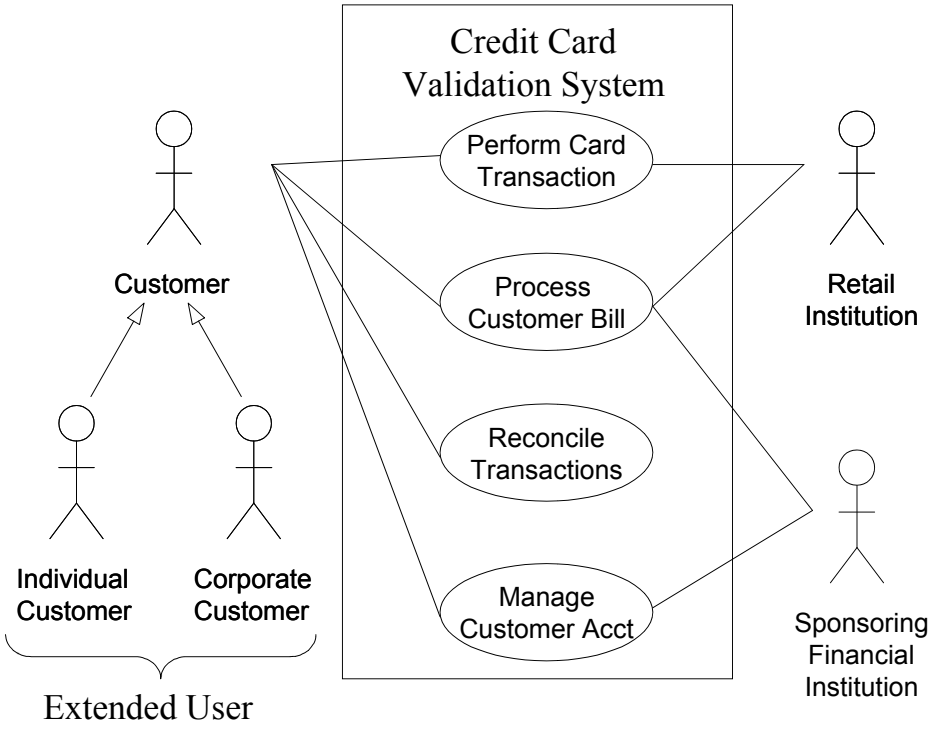


© Copyright Virtual University of Pakistan

The second diagram demonstrates the concept of reuse by extending already existing use cases. In this case *Place Conference Call* use case is a specialization of *Place Phone Call* use case. Similarly, *Receive Additional Call* is defined by extending *Receive Phone Call*. It may be noted here that, in this case, the arrow goes from the new use case that is being created (derived use case) towards the use case that is being extended (the base use case).

This diagram also demonstrates that many different actors can use one use case. Additionally, the actors defined for the base use case are also defined by default for the derived use case.

The concept of reusability can also be used in the case of actors. In this case, new classes of actors may be created by inheriting from the old classes of actors.



In this case two new classes, *Individual Customer* and *Corporate Customer*, are being created by extending *Customer*. In this case, all the use cases available to *Customer* would also be available to these two new actors.

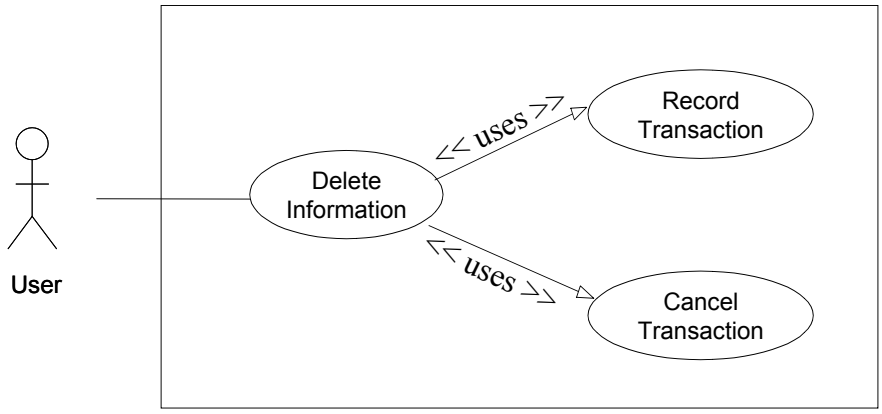
© Copyright Virtual University of Pakistan

3.10 Elaborated Use Cases

After the derivation of the use case model, each use is elaborated by adding detail of interaction between the user and the software system. An elaborated use case has the following components:

- Use Case Name
- Implementation Priority: the relative implementation priority of the use case.
- Actors: names of the actors that use this use case.
- Summary: a brief description of the use case.
- Precondition: the condition that must be met before the use case can be invoked.
- Post-Condition: the state of the system after completion of the use case.
- Extend: the use case it extends, if any.
- Uses: the use case it uses, if any.
- Normal Course of Events: sequence of actions in the case of normal use.
- Alternative Path: deviations from the normal course.
- Exception: course of action in the case of some exceptional condition.
- Assumption: all the assumptions that have been taken for this use case.

As an example, the *Delete Information* use case is elaborated as follows:



Use Case Name: Delete Information

Priority: 3

Actors: User

Summary: Deleting information allows the user to permanently remove information from the system. Deleting information is only possible when the information has not been used in the system.

Preconditions: Information was previously saved to the system and a user needs to permanently delete the information.

© Copyright Virtual University of Pakistan

Post-Conditions: The information is no longer available anywhere in the system.

Uses: Record Transactions, Cancel Action

Extends: None

Normal Course of Events:

1. **The use case starts when the user wants to delete an entire set of information such as a user, commission plan, or group.**
2. **The user selects the set of information that he/she would like to delete and directs the system to delete the information. - Exception 1, 2**
3. The system responds by asking the user to confirm deleting the information.
4. The user confirms deletion.
5. Alternative Path: Cancel Action
6. A system responds by deleting the information and notifying the user that the information was deleted from the system.
7. Uses: Record Transaction
8. This use case ends.

Alternative Path - The user does not confirm Deletion

1. If the user does not confirm deletion, the information does not delete.
2. Uses: Cancel Action

Exceptions:

1. The system will not allow a user to delete information that is being used in the system.
2. The system will not allow a user to delete another user that has subordinates.

Assumptions:

1. Deleting information covers a permanent deletion of an entire set of data such as a commission plan, user, group etc. Deleting a portion of an entire set constitutes modifying the set of data.
2. Deleted information is not retained in the system.
3. A user can only delete information that has not been used in the system.

3.11 Alternative Ways of Documenting the Use Case

Many people and organizations prefer to document the steps of interaction between the use and the system in two separate columns as shown below.

User Action	System Reaction
1. The use case starts when the user wants to delete an entire set of information such as a user, commission plan, or group.	

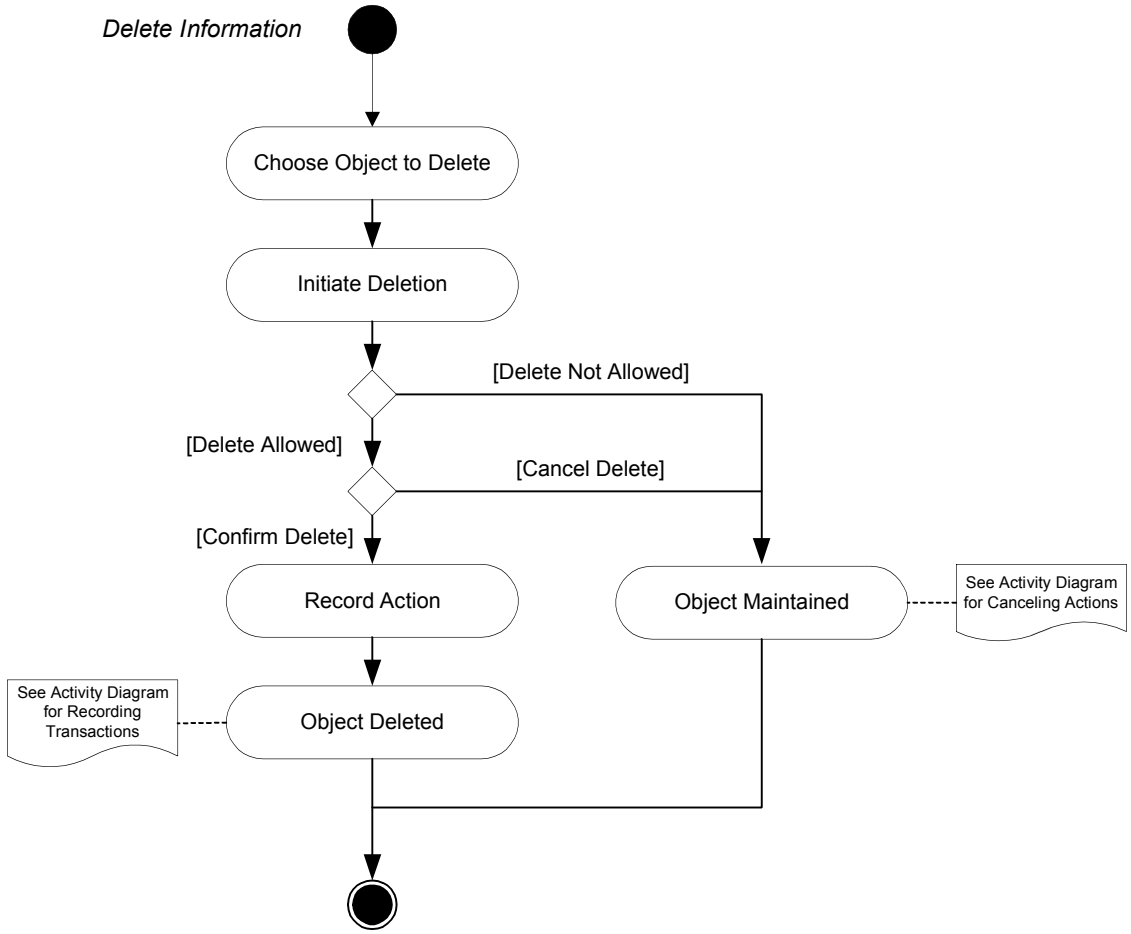
© Copyright Virtual University of Pakistan

2. The user selects the set of information that he/she would like to delete and directs the system to delete the information. - Exception 1, 2	3. The system responds by asking the user to confirm deleting the information.
4. The user confirms deletion.	**5. A system responds by deleting the information and notifying the user that the information was deleted from the system.**

It is a matter of personal and organizational preference. The important thing is to write the use case in proper detail.

3.12 Activity Diagrams

Activity diagrams give a pictorial description of the use case. It is similar to a flow chart and shows a flow from activity to activity. It expresses the dynamic aspect of the system. Following is the activity diagram for the *Delete Information* use case.



© Copyright Virtual University of Pakistan

3.13 Limitations of Use Cases

Use cases alone are not sufficient. There are kinds of requirements (mostly non-functional) that need to be understood. Since use cases provide a user's perspective, they describe the system as a black box and hide the internal details from the users. Hence, in a use case, domain (business) rules as well as legal issues are not documented.

The non-functional requirements are also not documented in the use cases. As examples of those, consider the following requirements.

- Usability
  - Color blind people should not have any difficulty in using the system – color coding should take care of common forms of color blindness.
- Reliability
  - The system needs to support 7 x 24 operation
- Performance
  - Authorization should be completed within 1 minute 90% of the time.
  - Average authorization confirmation time should not exceed 30 seconds.
- Portability
  - The system should run on Windows 98 and above as well as Sun Solaris 7.0 and above.
- Access
  - System should be accessible over the internet – hidden requirement – security

Because of this shortcoming, use cases must be augmented by additional information.

© Copyright Virtual University of Pakistan

Lecture No. 7

3.14 Source and sink analysis

Once requirements are documented using any of these analysis models, an independent verification is needed to verify completeness and consistency of requirements captured through these models. The process of verifying requirements involves careful analysis of sources as well as the sinks of information.

Source

A stakeholder describes requirements (needs, constraints) to be included as system functionality. These can be processes that generate certain information that the system may have to process or maintain. Sources of requirements are the origins from where the corresponding business process is initiated. By this concept, one has to trace from a requirement back to its origins to see who is involved in its initiation. Be it a person, an organization or an external entity that initiate some action and system responds back by completing that action.

Sink

Sink is the consumer of certain information. It is that entity which provides a logical end to a business process. Thus, 'sinks of requirements' is a concept that helps in identifying persons, organizations or external systems that gets certain functionality from the system. These are logical ends of requirements, or where all the requirements are consumed. For example, we may consider a user of a software application that retrieves a report from the system. In this case, user when reviews the report, becomes the sink of that report. Thus when analyzing the sink of the requirement of implementing a report, the analyst would naturally point towards the user who would get that report.

In source and sink analysis the analyst determines all the sources of requirements and where do these requirements consume (sinks). Now evaluate a report which displays certain information, the source of this report is the data (and who enters it) that is input to be retrieved later in the form of the report. Similarly, whoever needs this report become the sink of the report.
In a similar manner, at times we gather data in our application that is not used anywhere. So the question really is what to do with that kind of unused data or the missing requirement. Is it really redundant or is something really missing from these requirements? How to figure it out?

For example, we are having certain inputs (sources) to a process against which we do not know about the corresponding outputs (sinks). Such inputs are redundant if there is found no corresponding outputs. Thus these inputs can be removed as redundant. If we probe out corresponding outputs, which could not be recorded initially, that mean these inputs were not redundant rather a few (output related) requirements were missing that we discovered during the sink analysis.

© Copyright Virtual University of Pakistan

A stakeholder may have required the development team to develop certain report for his use. It means we are sure of its use (sink) but not about its sources, from where the required information will be provided? Who will input that information and using what mechanism?

A requirement statement that describe the report but do not list down its sources, will be an incomplete statement and the software engineer who is involved in validating such requirements, should identify all the sources against sinks or vice versa to determine complete end-to-end requirements.

Process Models

Domain Models

During requirements analysis phase, different models are developed to express requirements of the system. Though it is difficult to draw a line between these models as they complement each other, they differ in the manner information is expressed in these models. Most of these models are pictorial and contain explanation to the diagrams. Some of these models are discussed in the following subsections.

Understanding the business domain

It must always be kept in mind that the first step in delivering a system is establishing what needs to be driven. That is, clear understanding of the problem domain is imperative in successful delivery of a software solution. A software developer has to develop an understanding of the business problem he is trying to solve. Unless he develops this understanding, it is really difficult, if not impossible, to develop the right solution. But at least if he collects both ends (sources, sinks) involved in different processes of the business system, the corresponding requirements will be complete and yield a better understanding of the problem domain. A software engineer works on domains that may not correspond to his field of specialization (computer science, software engineering). He may be involved in the development of an embedded application that automates the control pad of a microwave machine or a decision support application for a stock exchange broker. As the underlying systems for which these software applications are being developed are not software systems, the software engineer cannot be expected to know about these domains. So, how should he get all the required knowledge about these systems? As without acquiring this knowledge, he may not be able to write down complete and unambiguous requirements which are acceptable to users as well.

An important difference between software and another engineering discipline is that the software engineer has to work on problems that do not directly relate to software engineering. Whereas, an electrical engineer will work on electrical domain problems, a civil engineer will work on civil engineering problems and so on. So, software engineer has to learn user vocabulary and terms which they use in their routine operations. To overcome this problem, a number of domain gathering techniques are used. These techniques help in extracting requirements from systems which are not known to a software engineer. Using these techniques the requirements gathering and validation process becomes convenient and manageable for a software engineer.

The following subsections discuss some of these techniques.

4.2 Logical System Models

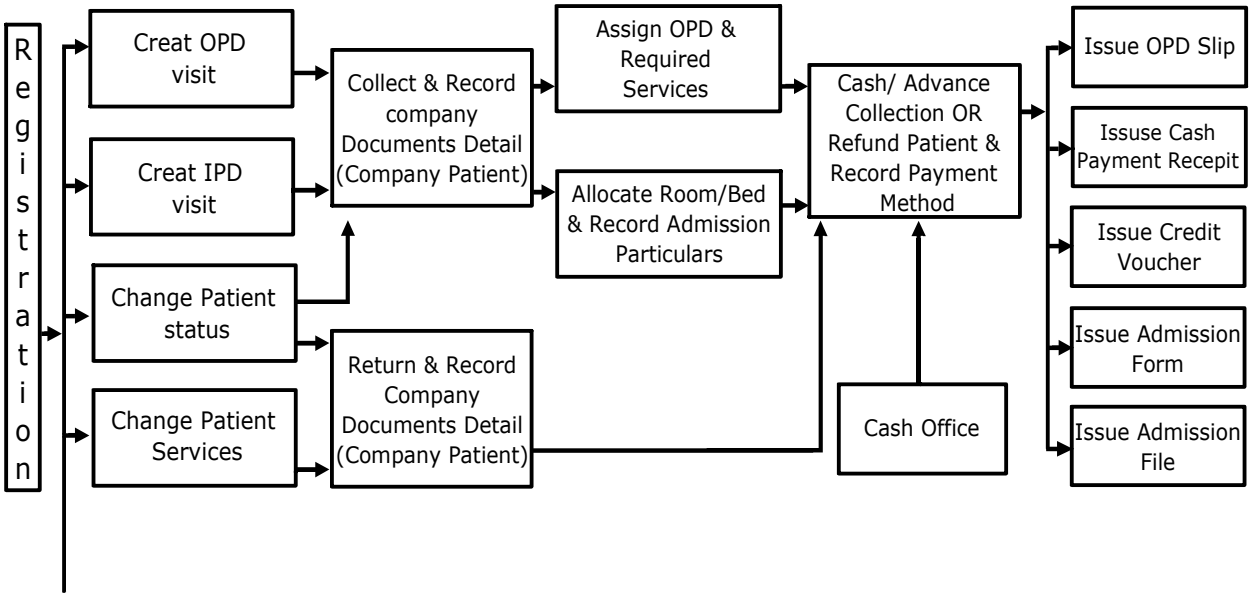
© Copyright Virtual University of Pakistan

System models are techniques used to understand user needs and software engineer use these techniques in order to understand business domain. Software engineers develop diagrams to model different business processes. System models include the following

- User business processes
- User activities for conducting the business processes
- Processes that need to be automated
- Processes which are not to be automated

Business process model

The first model that we will look at is called the process model. This model provides a high-level pictorial view of the business process. This model can be used as a starting point in giving the basic orientation to the reader of the document. Following is an example of a hospital registration system which deals with two types of patients.



As opposed to flow charts, there are parallel activities in this diagram which are further elaborated by specifying their major activities. The process described in this diagram is as follows

- A patient may come to visit In Patient Department (IPD) or output patient department (OPD)
- System determines if he is a company patient or a private patient.
- For a company patient, system verifies him.
- For an OPD patient, system will issue a chit to the patient and inform him about his number and the consultant to whom he has to consult and he will have to wait for his turn.
- After verifying an IPD patient, system will create a visit and allocate him a room or a bed etc. If system cannot allocate this, then it will inform the patient. Otherwise the patient is checked in and his information is maintained in the system.

© Copyright Virtual University of Pakistan

- System displays information about the expenses of the required service to the patient so that he is informed of his expected expenditure.
- Some advance payment is also received against the required service and this amount is adjusted in the final settlement.
- All this information is supplied to cash office that eventually deals with payments, etc.
- Upon receiving the cash, for OPD patient, a chit will be issued. For IPD patient, an admission form will be filled and this information will be maintained in the system. A receipt will be issued to the patient.
- For credit transaction, corresponding voucher will be prepared.
- So the model depicts process before the start of the treatment.
- A patient may ask to change his service on event of an unsatisfied response from the hospital staff or any other reason. System may cancel his record and pay his amount back.
- Similarly, a doctor may ask a patient to change his status from OPD to IPD.

In a business process diagram, following points are important and should be noted

- It does not describe the automated system
- It only reflects the existing process of the user to help software engineer/analyst in understanding business domain.
- It may contain information on processes that need not be automated.

© Copyright Virtual University of Pakistan

Lecture No.8

State Transition Diagrams

State transition diagrams (STDs) are another technique to document domain knowledge. In many cases, information flows from one place to the other and at each place certain action is taken on that piece of information before it moves to the next place. A file in an office is a typical office is example of such system. In this case, different people make comments and add information to that file and it moves from one table to the other this movement is controlled by a pre-defined set of rules which define under what condition the file moves from place A to place B and so on. We can easily document these set of rules with the help of state transition diagrams.

Following is an example of a use of STD to document the life cycle of a trouble ticket (this example has been taken from ITU-X.790 document).

A Trouble report and its life cycle – and introduction

From time to time all systems, including communications networks, develop problems or malfunctions referred to in this Recommendation as "troubles". A "trouble" in a communications network is a problem that has an adverse effect on the quality of service perceived by network users. When a trouble is detected, possibly as a result of an alarm report, a trouble report may be entered by a user or the system may raise a report automatically. Management of that trouble report is necessary to ensure that it receives attention and that the trouble is cleared to restore the service to its previous level of capability.

At the time of a trouble, a network may have been inter-working with another network to provide a service, and the problem or malfunction may be due to the other network. Therefore it may be necessary to exchange trouble management information between management systems across interfaces which may be client to service provider or service provider to service provider interfaces and may represent inter-jurisdictional as well as intra-jurisdictional boundaries. In addition to exchanging information on trouble that has already been detected, advance information on service inaccessibility may also need to be exchanged. Thus, a service provider may need to inform a customer of future service inaccessibility (because of planned maintenance, for example).

Trouble report states and status

Referring to the State transition diagram in Figure 2, a trouble report may go through any of six states during its life cycle. In addition, a T
rouble Status attribute is defined which qualifies the state (finer granularity) e.g. cleared awaiting customer verification. The time at which the status attribute change is also captured in the trouble report

.

Following is a description of states of a trouble report.

© Copyright Virtual University of Pakistan

Queued
A trouble report is in a queued state when it has been instantiated but the trouble resolution process has not yet been initiated. A trouble report which is in the queued state may be cancelled by the manager. The agent on receiving such a request will attempt to close the trouble report.

Open/active
The trouble report becomes "open/active" when appropriate actions to resolve the trouble are initiated.
An "open/active" trouble report may be "referred" to another Hand-off Person, or "transferred" to another Responsible Person for further processing. The state however remains unchanged as "open/active". A trouble report in the open/active state may be cancelled by the manager. The agent on receiving such a request will attempt to close the trouble report.

Deferred
This state indicates that corrective action to resolve the trouble has been postponed. This can occur when the faulty resource is inaccessible for a period and repair activity cannot proceed. A deferred Telecommunications Trouble Report may become "open/active" again, or move directly to the "closed" state if it is cancelled for some reason. A trouble report in the deferred state may be cancelled by the manager. The agent on receiving such a request will attempt to close the trouble report.

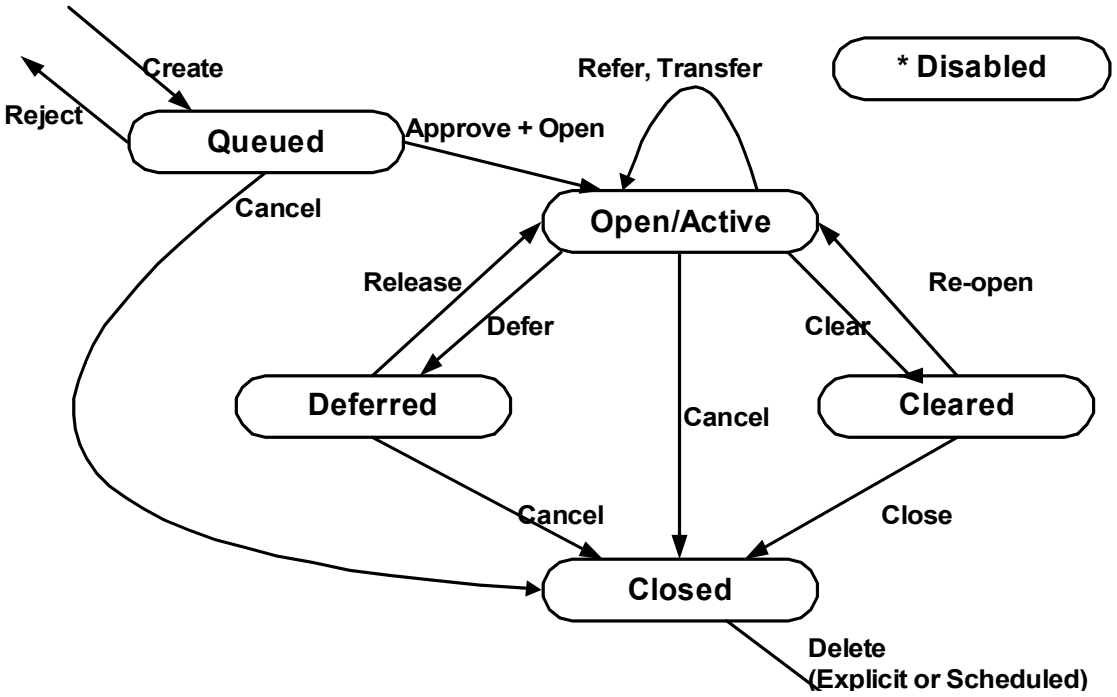
Cleared
A trouble report is moved by the agent to the "cleared" state when it determines that the trouble has been resolved. If the manager needs to verify that the trouble has been resolved, verification may optionally be awaited by the agent prior to closure of the trouble report.

Closed
This state indicates that the trouble resolution process is complete. Upon closure, the trouble report attributes are captured in a historical event generated at trouble report closure which may then be stored in a log of trouble history records, for future reference. The trouble report may then be eliminated at the agent's convenience. However, the agent may be required to maintain such records for a period of time as per business agreements.

Disabled
A "disabled" value is exhibited when a trouble report's information cannot be updated due to local conditions. In the "disabled" condition only read operations can be performed.

The following figure shows the STD for a trouble ticket. This diagram depicts the movement of a trouble ticket from one state to the other, thus making it easy to understand.

© Copyright Virtual University of Pakistan



© Copyright Virtual University of Pakistan

Arranging information in tabular form

Sometimes it is better and more convenient to arrange information in a tabular form. This makes it easier for the reader to understand and comprehend the information and hence designing, coding, and testing become less challenging. As an example, let us look at the following definitions used for identifying different data functions in the function point analysis taken from International Function Point User's Group (IFPUG) Counting Practices Manual (CPM 4.1).

External Inputs

An external input (EI) is an elementary process that processes data or control information that comes from outside the application boundary. The primary intent of an EI is to maintain one or more ILFs and/or to alter the behavior of the system.

External Outputs

An external output (EO) is an elementary process that sends data or control information outside the application boundary. The primary intent of an external output is to present information to a user through processing logic other than, or in addition to, the retrieval of data or control information. The processing logic must contain at least one mathematical formula or calculation, or create derived data. An external output may also maintain one or more ILFs and/or alter the behavior of the system.

External Inquiry

An external inquiry (EQ) is an elementary process that sends data or control information outside the application boundary. The primary intent of an external inquiry is to present information to a user through the retrieval of data or control information from an ILF or EIF. The processing logic contains no mathematical formulas or calculations, and creates no derived data. No ILF is maintained during the processing, nor is the behavior of the system altered.

It is difficult to understand these definitions and one has to read them a number of times to understand what is the difference between EI, EO, and EQ and in which case a function would be classified as EI, EO, or EQ.

Now the same information is presented in the tabular form as follows:

Function	Transactional Function Types		
	EI	EO	EQ
Alter the behaviour of the system	PI	M	N/A
Maintain one or more ILFs	PI	M	N/A
Present information to the user	M	PI	PI

PI – Primary intent; M – may be; N/A – not allowed.

This table simply says that a function can alter the behaviour of the system, it can maintain one or more ILFs, and/or it can present information to the user. The next step is to determine whether it is EI, EO, or EQ. For that we have to determine what is the primary intent (PI) of the function and in addition to this primary intent, what else does it

© Copyright Virtual University of Pakistan

do. Identification of EQ is simple - in this case the only thing a function does is present information to the user, which is also it's primary intent. If it alters the behaviour of the system or maintains and ILF then it can either be an EI or and EO but not an EQ. On the other hand if the primary intent of the function is to present information to the user but at the same time it also performs any of the first two operations, it is an EO. Finally, if the primary intent of the function is either to alter the behaviour of the system of maintain one or more ILFs, then it is an EI.

Hence by putting and organizing the information in the form of a table, we have not only made it simple to understand the definition but also given an holistic picture which was not easily visible otherwise.

Let us look at another example. This time the information is taken from the Income Tax Ordinance of Pakistan 2001. Consider the following statement that describes the income tax rates applicable to people with different brackets:

If the taxable income is less than Rs. 60,000, there will be no income tax. If the income exceeds Rs. 60,000 but is less than Rs. 150,000 then income tax will be charged at the rate of 7.5% for income exceeding Rs. 60,000. If the income exceeds Rs. 150,000 but does not exceed Rs. 300,000 then the income tax will be computed at 12.5% of the amount exceeding Rs. 150,000 plus Rs. 6,750. If the income exceeds Rs. 300,000 but does not exceed Rs. 400,000 then the income tax will be computed at 20% of the amount exceeding Rs. 300,000 plus Rs. 25,500. If the income exceeds Rs. 400,000 by does not exceed Rs. 700,000 then the income tax will be computed at 25% of the amount exceeding Rs. 400,000 plus Rs. 45,500. If the income exceeds Rs. 700,000 then the income tax will be computed at 35% of the amount exceeding Rs. 700,000 plus Rs. 120,500.

The same information can be organized in the form of a table, making it more readable and easier to use.

Income	Tax
Less than Rs. 60,000	0%
Between Rs. 60,000 and Rs. 150,000	7.5% of (Income - 60,000)
Between Rs. 150,000 and Rs. 300,000	12.5% of (Income - 150,000) + 6,750
Between Rs. 300,000 and Rs. 400,000	20% of (Income - 300,000) + 25,500
Between Rs. 400,000 and Rs. 700,000	25% of (Income - 400,000) + 45,500
Greater than Rs. 700,000	35% of (Income - 700,000) + 120,500

Once the information has been organized in the tabular form, in many cases it can be simply stored and mapped onto an array or a database table and the programming of this

© Copyright Virtual University of Pakistan

kind of a rule is simply reduced to a table or dictionary lookup. This reduces the complexity of the domain and hence reduces the over all effort for designing, coding, testing, and maintaining the system.

Data Flow Model

- Captures the flow of data in a system.
- It helps in developing an understanding of system's functionality.
- What are the different sources of data, what different transformations take place on data and what are final outputs generated by these transformations.
- It describes data origination, transformations and consumption in a system.
- Information is organized and disseminated at different levels of abstraction. Thus this technique becomes a conduit for top down system analysis and requirements modeling.

The Notation

There are several notations of the data flow diagrams. In the following, four different shapes are explained.

Process

- What are different processes or work to be done in the system.
- Transforms of data.



External Agent

External systems which are outside the boundary of this system. These are represented using the squares



Data Store

- Where data is being stored for later retrieval.
- Provides input to the process
- Outputs of the processes may be going into these data stores.

© Copyright Virtual University of Pakistan



Data Flow

- Where the data is flowing.
- Represents the movement of the data in a data flow diagram.



DFD versus Flow Charts

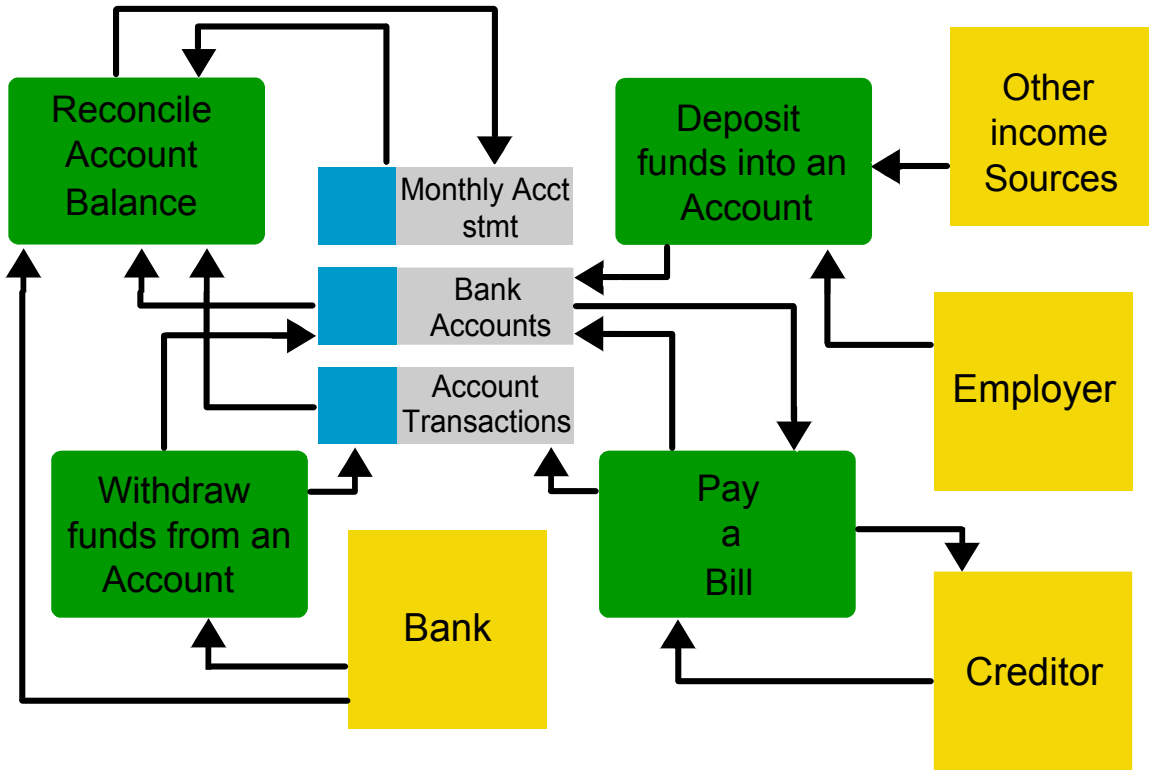
Flow charts are usually used to describe flow of control in a system. It describes control flow in an algorithm. Flow charts are quite detailed. Whereas DFD does not captures control flow information, it just shows the flow of the data in a system. Flow charts show the sequential activities of an algorithm. So, decisions are made, loops or iterations are described. On the other hand, DFD does not show the sequential activities. It just displays the business flow (without sequence among activities). As if you visit an organization, business activities are being performed in parallel. Therefore, DFD does not contain control or sequential activities just data transition is captured.

DFD	Flow Chart
• Processes on a data flow can operate in parallel. • Looping and branching are typically not shown. • Each process path may have a very different timing.	• Processes on flowcharts are sequential. • Show the sequence of steps as an algorithm and hence looping and branching are part of flowcharts.

Data Flow Model – Bank Account Management System

In the following, we are presenting a data flow model that describes an accounts management system for a bank. This data flow diagram consists of the following entities

© Copyright Virtual University of Pakistan



Processes

1. Reconcile account balance
2. Deposit funds into an account
3. Pay a bill
4. Withdraw funds from an account

External agents

1. Bank
2. Creditor
3. Employer
4. Other income sources

Data stores

1. Monthly Account statement
2. Bank accounts
3. Account transactions

Description:

First we shall discuss 'withdraw funds from an account' process. In this process, information about the accounts and account transactions is retrieved (from the data stores) and bank releases the funds. After this, it sends this information to 'reconcile account balance' process which prepares a monthly account statement. In this statement, information regarding bank accounts and account transactions are described. Next is the 'pay a bill' process through which a creditor pays his dues and the corresponding accounts are updated against the cash transaction. A receipt is issued back to the creditor. The fourth process is 'deposits funds in an account' in which an employer deposits salaries of his employees and the salary information is deposited in the corresponding

© Copyright Virtual University of Pakistan

bank accounts of the employees. Similarly, income received through other income sources is also received and deposited in the corresponding bank accounts.

Data Flow Modeling

When data flow modeling is used to model a system's functionality, following points need to be remembered

- Data flow model captures the transformation of data between processes/functions of a system. It does not represent the control flow information that is occurring in a system to invoke certain functionality.
- A number of parallel activities are shown in this diagram where no specific sequence among these activities is depicted
- All the previous models that we studied like business process models, state transition diagrams, are used to capture business domain irrespective of their automation.
- However, in data flow models, we represent only those processes which we need to automate as they involve certain computation, processing or transformation of data that can be best implemented using an automated system.
- For example, we may consider a mail desk in an office that receives mail and just forwards it to their respective addressees. In this example, as the mail desk does not process the mail, just forwards it, therefore it does not include any process that need to be automated. Hence, we shall not use data flow diagrams to model this process.
- In nutshell, processes that just move or transfer data (do not perform any processing on that data), should not be described using data flow models.
- Taking the same example, if we modify the scenario such that a mail desk clerk receives the mail, notes it down into a register and then delivers it to their respective addressees then a processing has got involved in this scenario. At least one process is there that can be automated. That is, the recording of mail information into the register. Now we can use a data flow model in which we shall use a data transformation that captures the detail of recording mail information into a register (or a data store). Thus with this addition, it makes sense to use data flow model to capture the details of this process.

© Copyright Virtual University of Pakistan

Lecture No. 9

4.3 Typical Processes

Now we shall discuss processes which are typically modeled using data flow diagrams. These processes transform data in one or the other way but these are found in almost all the automated systems. Following are the examples

- Processes that take inputs and perform certain computations. For example, Calculate Commission is a process that takes a few inputs like transaction amount, transaction type, etc and calculates the commission on the deal.
- Processes which are involved in some sort of decision-making. For example, in a point of sales application a process may be invoked that determines the availability of a product by evaluating existing stocks in the inventory.
- Processes that alter information or apply a filter on data in a database.

For example , an organization is maintaining an issue log of the issues or complaints that their clients report. Now if they want to see issues which are outstanding for more then a weeks time then a filter would have to be applied to sort out all the issues with Pending status and whose initiation date is a week old.

- Processes that sort data and present the results to users. For example, we pass an array of arbitrary numbers to a QuickSort program and it returns an array that contains the sorted numbers.
- Processes that trigger some other function/process

For example, monthly billing that a utility company like WAPDA, PTCL generates. This is a trigger that invokes the billing application every month and it prepares and prints all the consumer bills.

- Actions performed on the stored data. These are called CRUD operations and described in the next subsection

CRUD Operations

These are four operations as describes below

- **Create**: creates data and stores it.
- **Read**: retrieves the stored data for viewing.
- **Update**: makes changes in an stored data.
- **Delete**: deletes an already stored data permanently.

4.4 Adding Levels of Abstraction to Data Flow Modeling

As we have already described that in data flow modeling only those processes can be expressed that perform certain processing or transformation of information. Now the question arises how far these processes need to be expressed? As a single process like CalculateCommission as described in the above section, can be described in sufficient detail such that all of its minute activities can be captured in the data flow diagram. However, if we start adding each bit of system functionality in a single data flow diagram, it would become an enormously large diagram to be drawn on a single piece of paper. Moreover, requirement analysis is an ongoing activity in which knowledge expands as you dig out details of processes. Therefore, it may not be possible for an

© Copyright Virtual University of Pakistan

analyst to know each bit of all the processes of the system from the very beginning. Keeping the complexity of systems in view, data flow modeling technique has suggested disseminating information of a system in more then just one levels of abstraction. What are these levels, please see below for a discussion

Context Level Data Flow Diagram

In a top-down system analysis, an analyst is required to develop high level view of the system at first. In data flow modeling, this high-level view is the Context level data flow diagram. In this diagram, system's context is clarified such that all the external agents or entities with which the system interacts are captured. It captures the details of what information flows between the system and these external entities, and what outputs are generated against inputs from these external agents and so on. So, the analyst probes out all the external agents that may involve persons, organizations or other systems who directly interacts with this system and their specific involvement in the system. At this level, systems internal details are not exposed, as we want to see system behavior as a black box.

Detailed Data Flow diagrams

Once context of a system has been captured using context level diagram, the analyst would expand his activities and start digging out system's internal details. Therefore, the same context level diagram is further expanded to include all major processes of the system that make up system functionality. So, instead of portraying system as a black box entity, the analyst would add processes that deal with the external agents and produces certain outputs. This is level one of a data flow model.

In level two of data flow model, instead of refining the previous levels further, we take one process from the level one diagram and expands it in a level two diagram. Hence, a level one diagram that depict the whole system, may be expanded to more then one level two diagrams each of which describes exactly one process in detail which were listed in level one diagram as simply an oval (process or transform).

This process may continue to any level of details as the analyst can conveniently captures. Where diagram at a specific level is a refinement of one of the processes listed in a previous level. By adding levels of abstraction to a data flow diagram, it becomes natural for a software engineer or a requirement analyst to readily express his knowledge about the system in an appropriate level of data flow model that corresponds only to a specific set of functionality.

It should be noted here that the number of external agents and their inputs to the system and the outputs that the system would return to them, should remain the same throughout different levels of a data flow model. It should be considered a mistake if context level diagram contains three external agents, which are providing two inputs each, and getting one output in return but at level one, we add one more external agent or input or the outputs. This would make level one model inconsistent with the context level diagram. This is true for any level of data flow model. For instance, at level two the number of external agents, inputs and outputs shown in (all of level two) diagrams should match exactly with the external agents, inputs and outputs shown in level one diagram. Therefore, disseminating information at an appropriate level of abstraction with the additional check of inter-level consistency makes data flow modeling a very powerful domain-modeling tool. After this discussion, we shall give the reader an example in

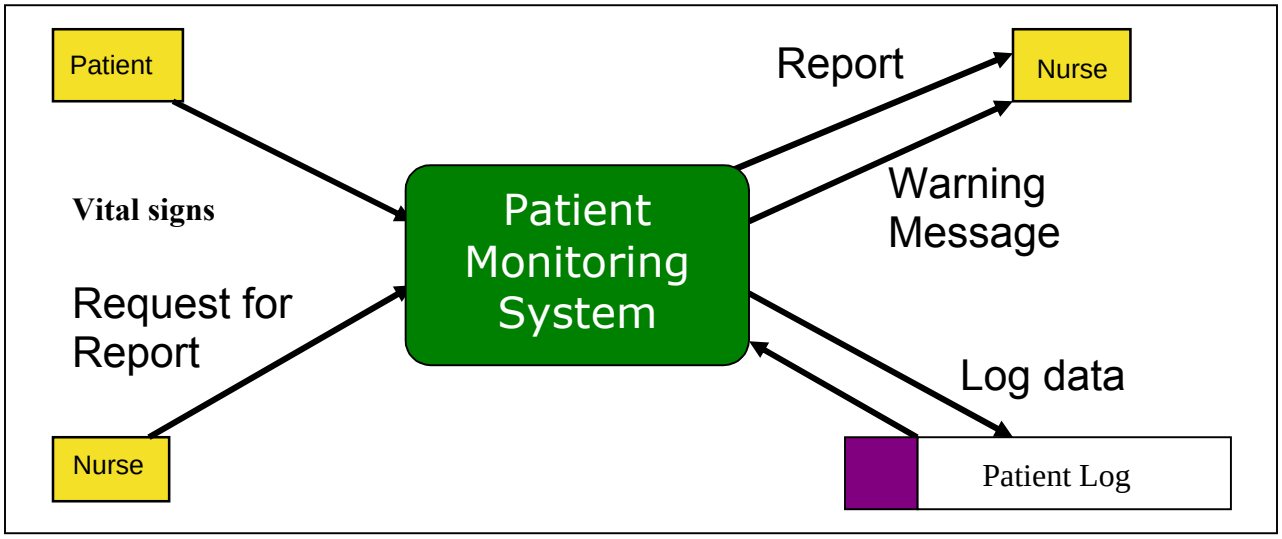
© Copyright Virtual University of Pakistan

which a system is modeled using data flow modeling technique where three levels of abstraction have been developed.

Patient Monitoring System – A Data Flow Modeling Example

Context Diagram

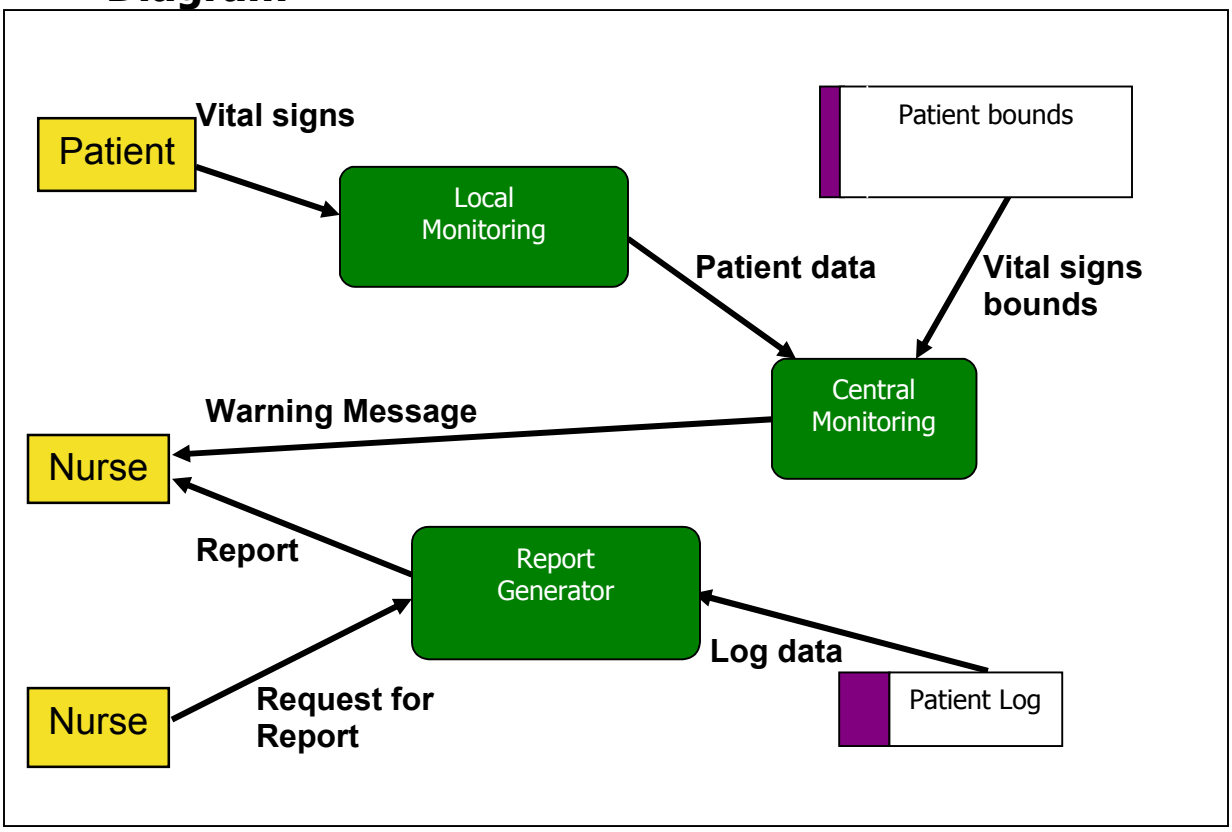
Following is the 0-level or the context level data flow diagram of the Patient Monitoring System. In this data flow diagram, three external entities (users) are interacting with the centralized system. Point to note here is that in this context level diagram, only one process or transform takes place that is the Patient Monitoring System itself. A patient's vital signs are transmitted to this system which may invoke a warning message to the nurse if these signs fall into the critical range. Nurse may request for a report, which the patient monitoring system retrieves from the patient log, and return it to the nurse again. In this manner the 0-level data flow diagram describes the context of this system.



In order to see detail processes involved in Patient Monitoring System, a level 1 data flow diagram will have to be made. In the following, we are providing level 1 data flow diagram which is a refinement of the level-0 data flow diagram.

© Copyright Virtual University of Pakistan

Patient Monitoring System – Level 1 Data Flow Diagram



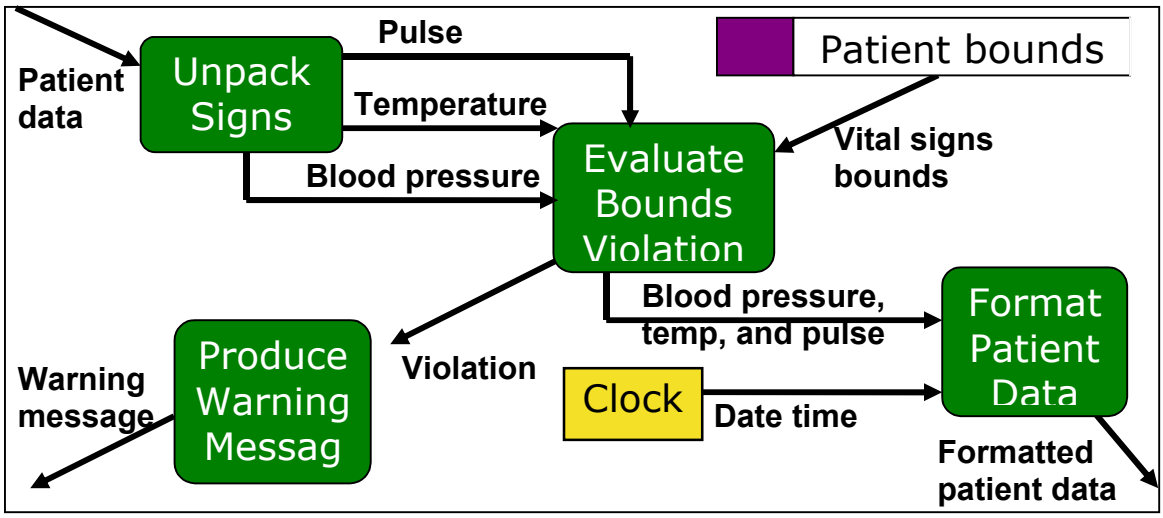
Level 1 data flow diagram is the refinement of the context (0-level) data flow diagram. All the external entities are the same (Nurse, and Patient), however, the process of 'Patient Monitoring System' is further elaborated by the three processes Local Monitoring, Report Generator, and Central Monitoring. The Local Monitoring process transforms vital signs that it receives from Patient entity into Patient data and passes this information to Central Monitoring process. Central Monitoring process retrieves vital signs bounds and compares Patient data and it may generate Warning message if the Patient data goes out of normal Vital signs bounds. A nurse may request for a report, in response the Report Generator process retrieves Log data from Patient Log, generates the report and displays it back to the nurse.

It should be noted here that this level 1 diagram is a further refinement of level 0 diagram such that the underlying system is the same but processes which were hidden in level 0 are represented in this diagram.

A further refinement of this model is also possible if we expand any of these three processes to capture further details. For example, the Local Monitoring process may further be expanded to capture detailed activities involved in the monitoring process. Following is level 2 diagram of Central Monitoring process.

Central Monitoring System – Level 2 Data Flow Diagram

© Copyright Virtual University of Pakistan



In the above level 2 data flow diagram, Patient's data is sent to the Unpack Signs process which unpacks it and send Pulse, Temperature, and Blood pressure to the Evaluate Bounds Violation process. This process retrieves Vital signs bounds information, compares it with unpacked patient data and sends a violation sign to the Produce Warning Message process upon an out of bound result of the comparison. The patient data is sent to Format Patient Data process that generates the formatted patient data to be maintained against patient profile. In this manner we elaborated the patient monitoring system up to three levels describing different details at each level. In a similar manner, other two processes in level 1 DFD could also be expanded in their respective level two diagrams in order to describe their functionality in more detail.

By going through this example, the reader would have learnt how data flow modeling technique helps in understanding domain of a system at different levels of abstractions. In the following sub-section, we shall describe common mistakes that the people do while preparing data flow diagrams.

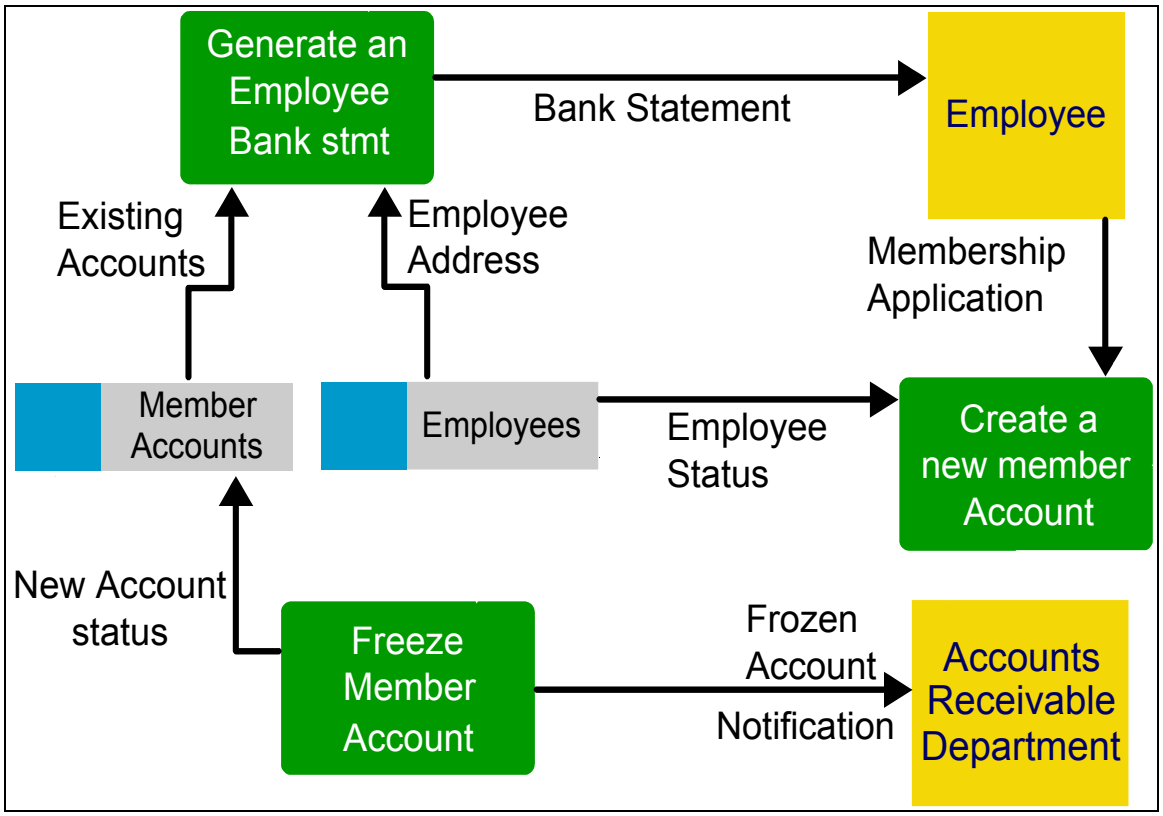
© Copyright Virtual University of Pakistan

4.5 Common Mistakes in Data Flow Diagrams

In the following data flow diagram, an accounting system has been described. Three processes are given Generate an Employee Bank Statement, Create a New Member Account, and Freeze Member Account. There are two external entities shown in this diagram Employee and Accounts Receivable Department. The three processes described in this diagram have associated problems. Can you guess these problems?

If you look at the arrows going inside each of these processes and coming out of them, you will observe some peculiarity.

In fact, here we can apply the source and sink analysis that we studied in the last lectures. What does the source and sink analysis suggest? It suggests that in order to check completeness of a requirement, evaluate the sources as well as the sinks of the requirements. Applying this knowledge in this case, we observe the following mistakes



- There is no input for the process Freeze Member Account
- In a similar manner, the process Create a New Member Account does not produce any output.
- Similarly, Generate Employee Bank Statement process is having two inputs and an output but the question really is, do these inputs correspond to the output?

The Freeze Member Account process that does not have any input is an example of a requirement whose source is not known. Similarly, the Create a New Member Account process that does not produce any output is an example of a requirement whose sink has not been specified. Lastly, the Generate an Employee Bank Account process though have two inputs and produces an output but in order to generate a bank statement, all that is

© Copyright Virtual University of Pakistan

needed is an account number and the time period for which the statement is required. If we analyze the inputs given to this process, we can observe that both of these inputs cannot help in generating the account statement. Therefore, these inputs are irrelevant to the output being generated by this process.

In the above mentioned example, it is evident that by applying the source and sink analysis we determined all the missing inputs and outputs to the processes of this diagram.

In the following subsection, we shall describe actions which are not only mistakes but illegal too for the data flow diagrams.

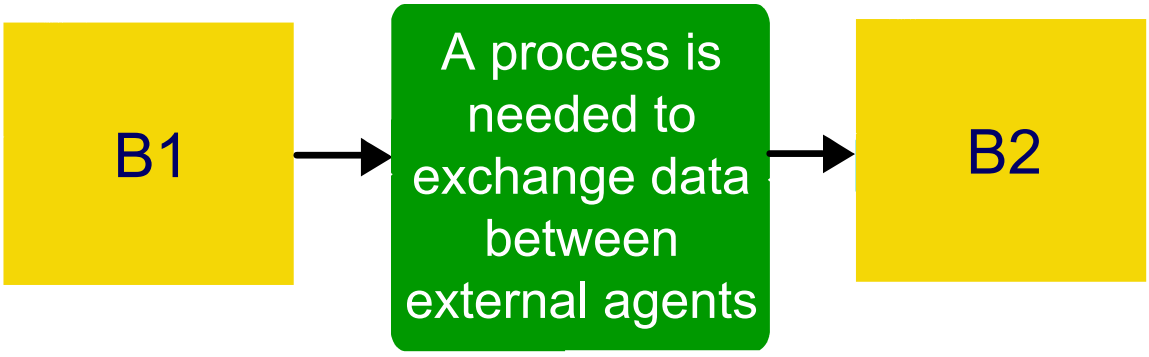
Illegal Data Flows

Directly Communicating External Agents

Following diagram depicts a scenario in which one external entity is directly communicating with another external entity. This form of communication is illegal to be shown in a data flow diagram.



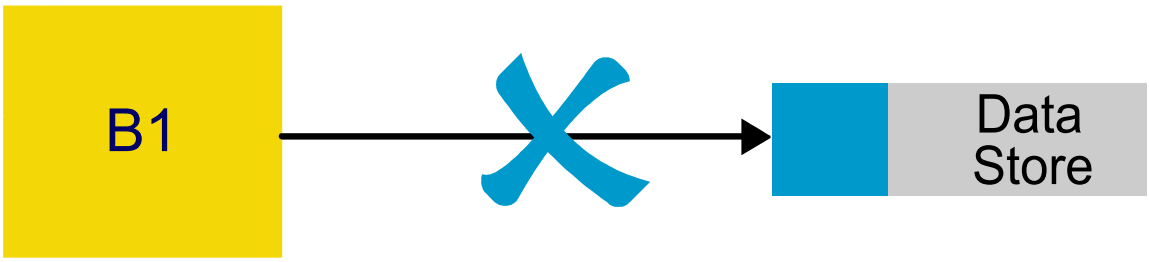
There must be an intermediate process which should transform data received from one external entity and then send the transformed data to the other external entity. As we have already described that data flow diagrams should be used to depict processes that transform or process data. Simple data movement from one entity to another should not be described using data flow diagrams.



External Agent updating information in a Data Store

© Copyright Virtual University of Pakistan

As we explained in the above case, a transform/process is needed between communicating entities. This is true even for an External Entity that wants to store/update some information directly in a data store, a transformation would be required.

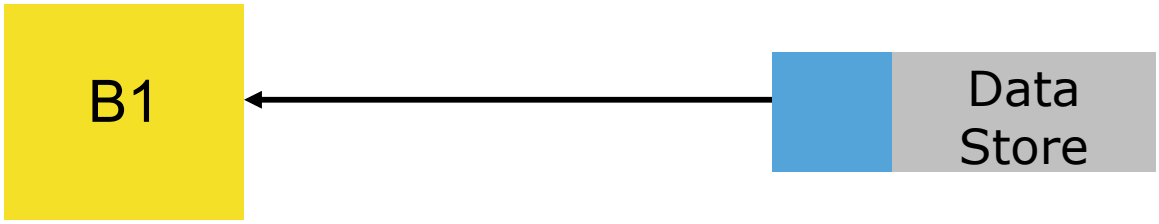


Therefore, a process should be inserted between the interacting entities (external agent, data store) that should store information received from the external agent after processing it.



External Agent accessing information from a Data Store

Similarly, an external agent accessing information from a data store directly is also illegal.



Again a data transform/process if needed in this communication. It should be able to retrieve information from the data store and then pass it on to the external agent.

© Copyright Virtual University of Pakistan

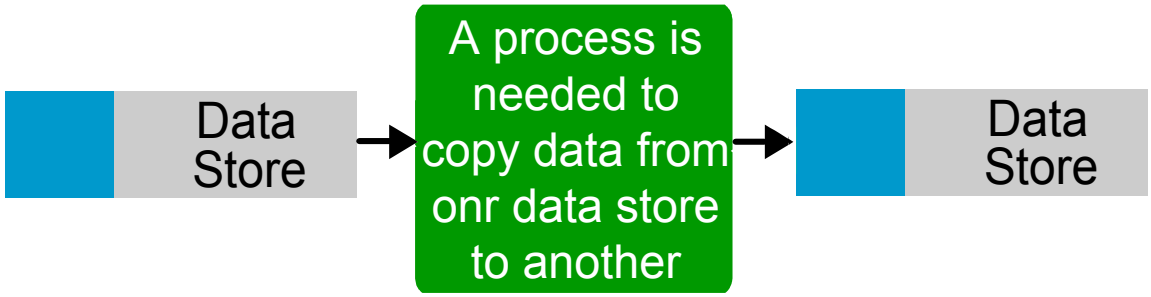


Copying data to a data store

In the following diagram, a data store is shown copying data directly to another data store. This is again illegal as there is not any intermediate process/data transform mentioned.



So, the correct method is again to use a data transform/process between the two data stores. It should retrieve data from one data store and after transforming that data, store it into another data store.



© Copyright Virtual University of Pakistan

Lecture No. 10

Prototyping and GUI Design

GUI Sketches

Adding user interface details in the SRS is controversial. The opponents of this argue that by adding GUI details to the SRS document, focus shifts from what to how – GUI is definitely part of the solution. On the other hand many people think that, it is still what not how and hence it should be made part of the SRS document. By adding the GUIs in the FS, requirements can be solidified with respect to scenario contents. It is my personal experience that the client appreciates more the contents of the SRS document if our SRS document contains the GUI details than if we don't have them there. This document is also going to be used as the base line for design, user manual, and test planning among other things. Presence of the UI details imply that these activities can start right after SRS is accepted and signed-off. Emergence of rapid GUI drafting tools has made the task a lot simpler than it used to be. Exploring potential user interfaces can be of help in refining the requirements and making the user-system interaction more tangible to both the user and the developer. User displays can help in project planning and estimation. A user interface might highlight weaknesses in addressing some of the non-functional requirements (such as usability), which are otherwise very hard to fix later on. If you cannot freeze the RS until UI is complete, requirement development process takes a longer time. However, we need to be very careful when we use GUIs to the SRS document. It is a very common mistake to use UI layouts as substitute of defining the functional requirements. We must remember that these are supplementary information and cannot replace other components of the SRS document. Any change in the requirements entails a change in the UI. If the requirements are not stable, this can mean a lot of rework.

Motivation for GUI

- System users often judge a system by its interface rather than its functionality
- A poorly designed interface can cause a user to make catastrophic errors
- Poor user interface design is the reason why so many software systems are never used

Pitfalls of using GUIs in Functional Specifications

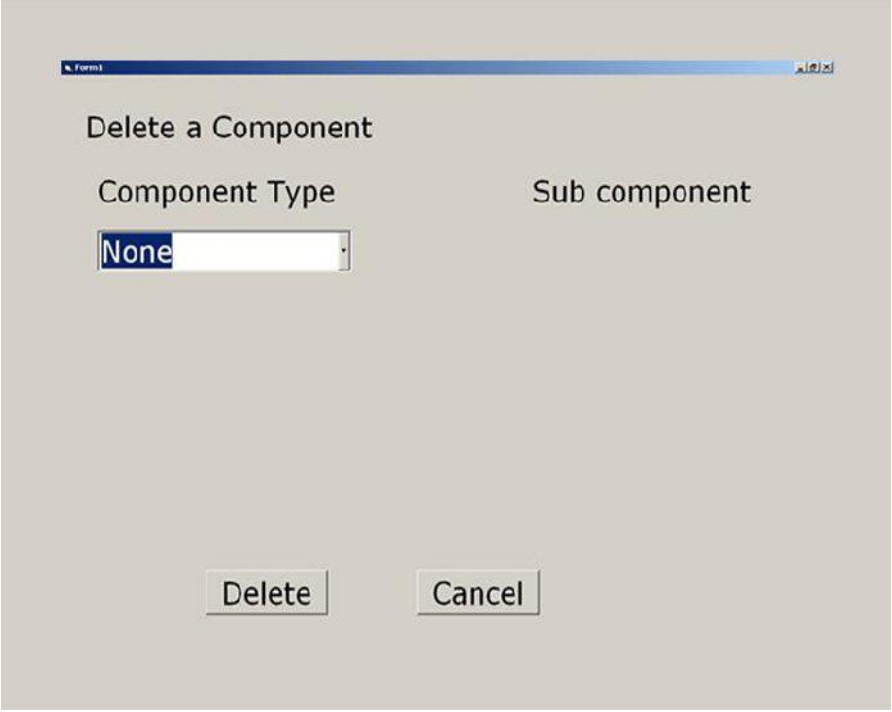
- UIs distract from business process understanding (what) to interfacing details (how)
- Unstable requirements cause frequent modifications in UIs
- An extra work to be done at the requirement level each time a GUI change has to be incorporated

In the following we shall discuss how unstable requirements cause difficulties in preparing GUIs

Example

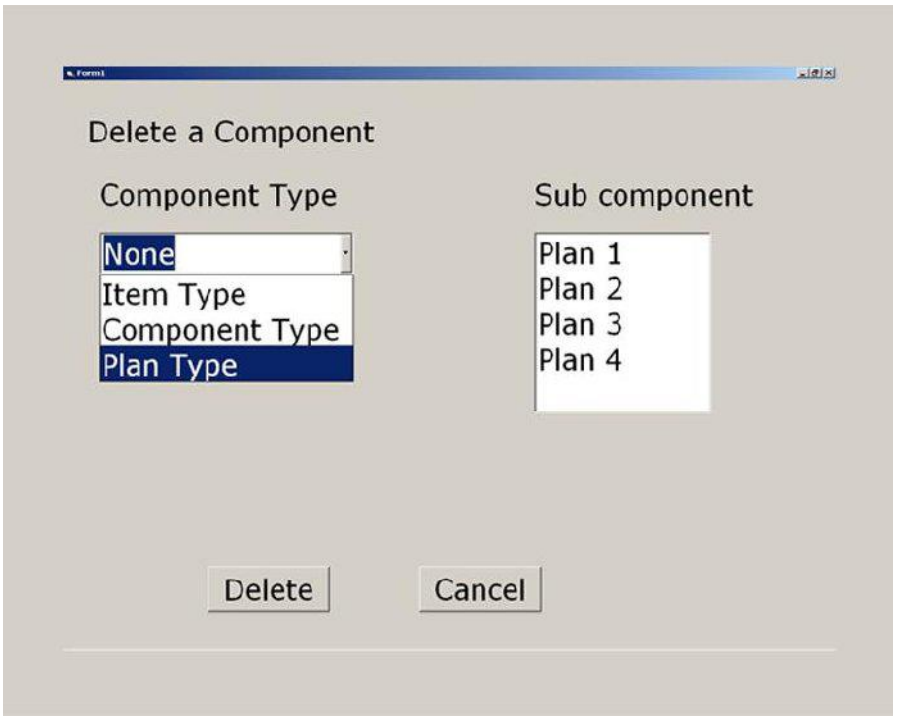
© Copyright Virtual University of Pakistan

The following GUI implements the delete component use case that we discussed in use case section. The GUI displays a drop down list box that contains a list of component types. The top of the list entry is 'None' where the user can click on the arrow and select the component type whose component he wants to delete.

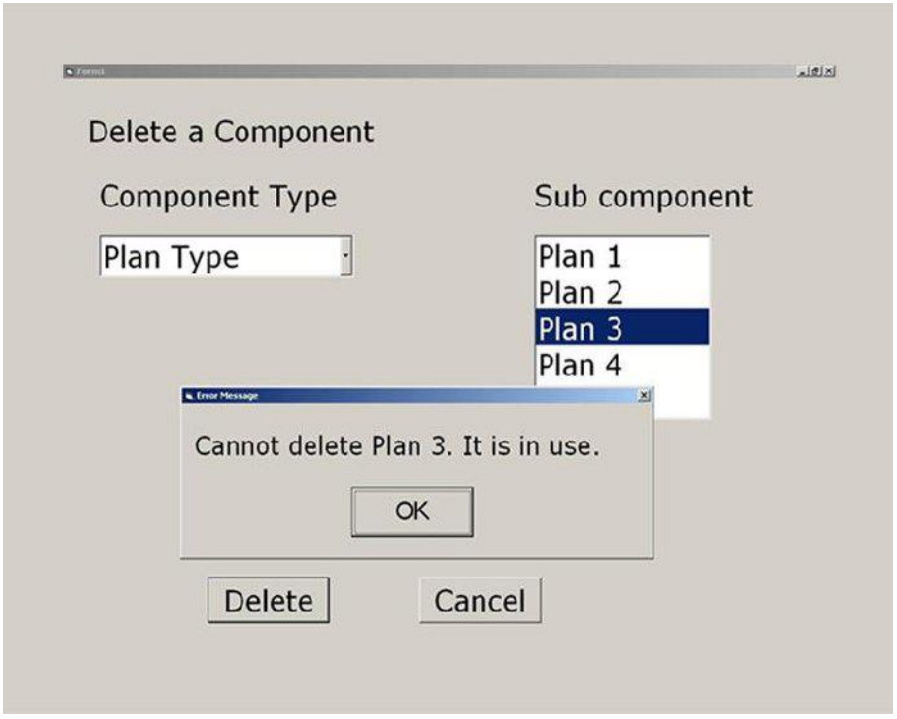


The next GUI implements the scenario when user has clicked over the arrow and a few component types are populated in the list. User then selects a component type 'Plan Type'. Corresponding plans are populated and displayed in the list box at the right side of the GUI.

© Copyright Virtual University of Pakistan

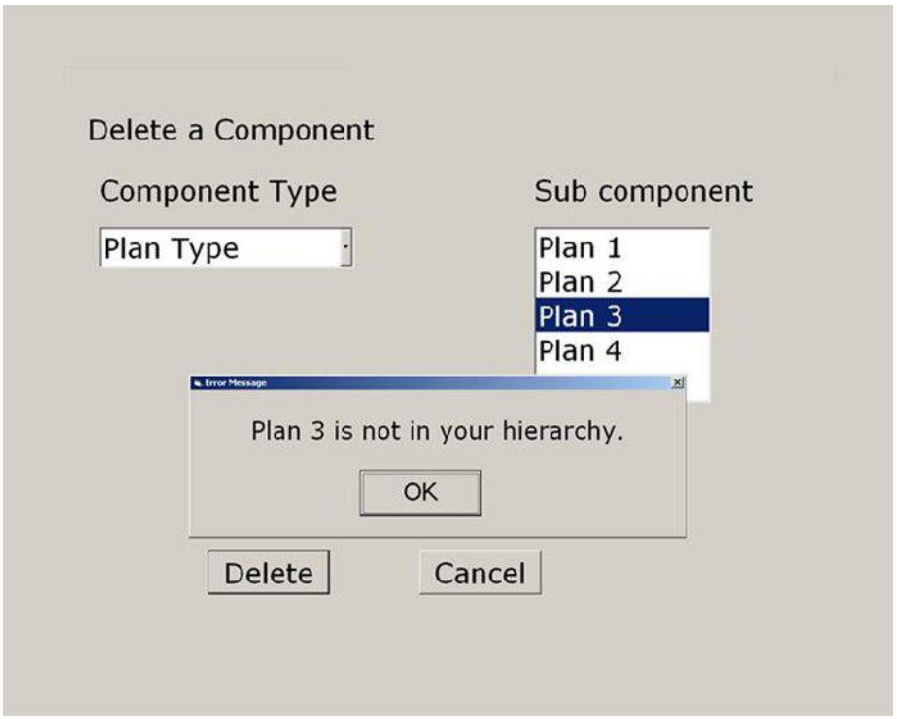


Following GUI depicts the scenario when user selects a particular plan 'Plan 3' and clicks on the 'Delete' button. Now assume that 'Plan 3' is currently being used. So, the application displays a dialog box to the user informing him that he cannot deletes this plan as it is in use.

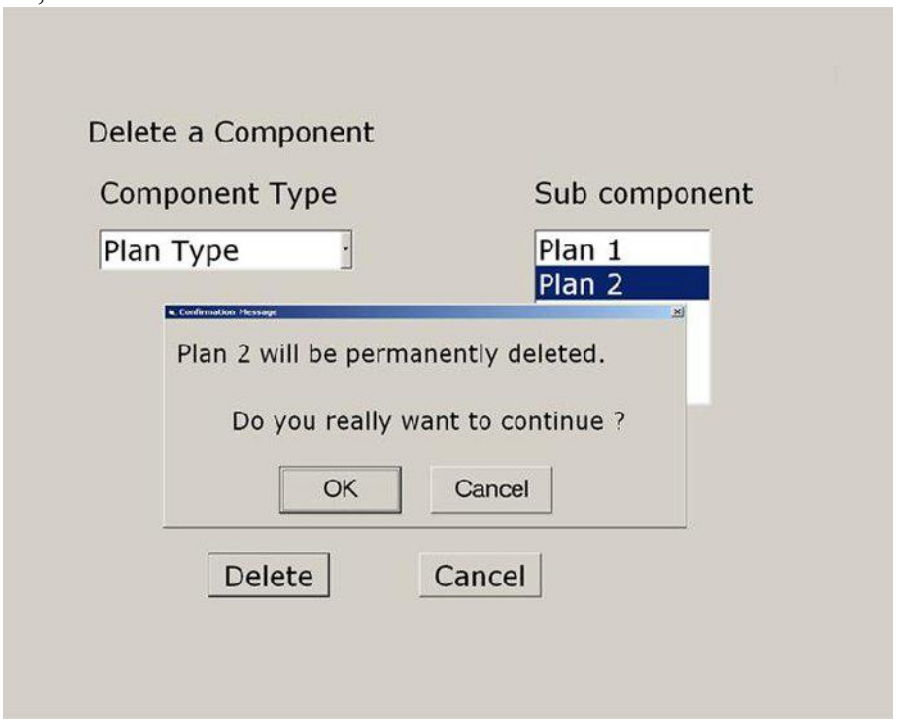


The next GUI, another dialog box is shown in which user is getting another message from the system. It says that Plan 3 is not in his hierarchy.

© Copyright Virtual University of Pakistan

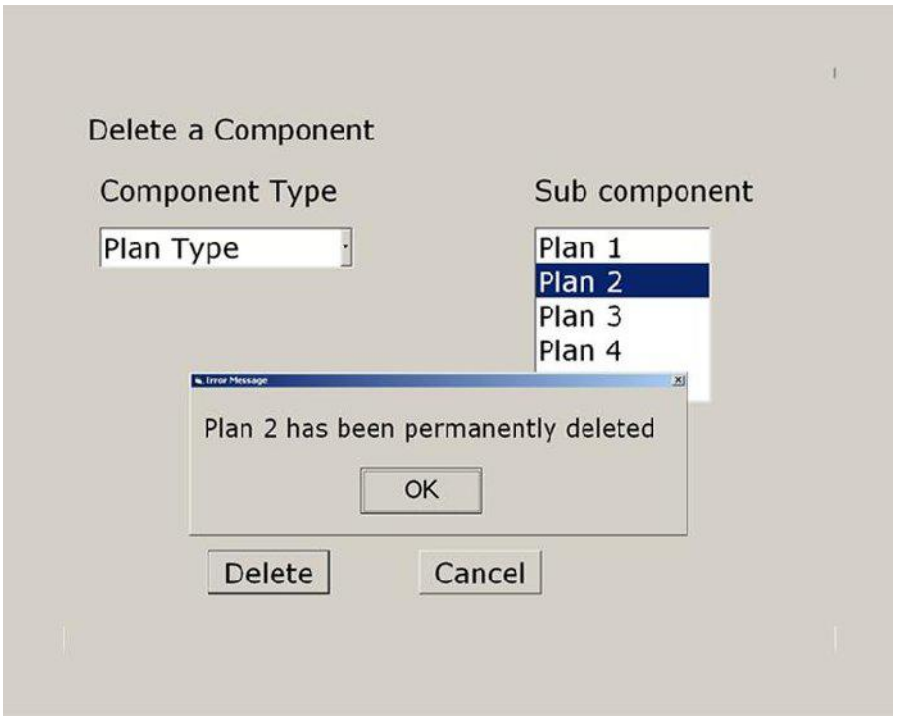


The user then selects 'Plan 2' and deletes it. System confirms the user and upon confirmation, deletes 'Plan 2'.

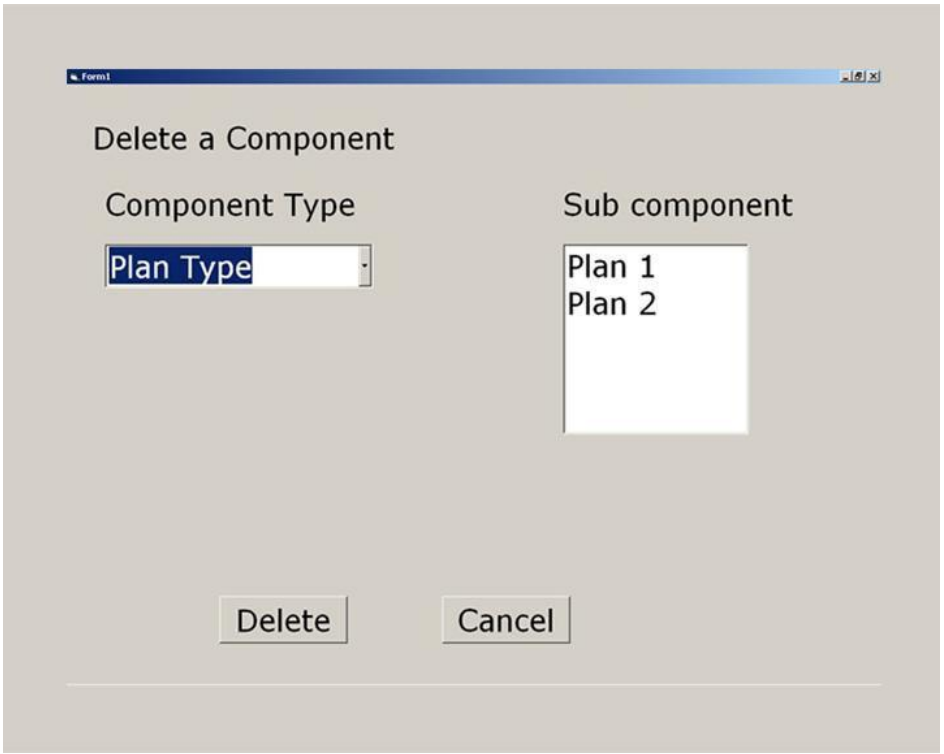


After deleting 'Plan 2', it displays the message that Plan 2 has been permanently deleted. Whereas, 'Plan 2' is still visible in the list.

© Copyright Virtual University of Pakistan



However, it should be noted that, all the above GUIs presented two major mistakes about the GUIs. First, if a plan is currently in use, it should not have been displayed in the list at the right. Secondly, instead of displaying two messages separately in two dialog boxes, it would have been appropriate to combine them in one message.
The following GUI displays what this GUI should have displayed ideally. As, user can only delete plans 1 and 2, therefore, only these plans should have displayed to him.



© Copyright Virtual University of Pakistan

In the above example, it is evident that if requirements are partially generated a number of changes have to be made and sometimes the frequency of these changes rise so much that it takes all of the requirements and design time just in finalizing GUIs.

© Copyright Virtual University of Pakistan

Prototype

Prototyping is yet another technique that can be used to reduce customer dissatisfaction at the requirement stage. The idea is to capture user's vision of the product and get early feedback from user to ensure that the development team understands requirements. This is used when there is uncertainty regarding requirements. Sometimes, even the customer does not know what he/she actually needs. This happens when there is no manual solution.

A prototype is not the real product. It is rather just a real looking mock-up of what would be eventually delivered and might not do anything useful. However, the presence of a prototype makes a new product tangible. It brings use cases to life and closes gaps in your understanding of the requirements. From a user's perspective, it is easier to play with a prototype and try it out than to read SRS.

© Copyright Virtual University of Pakistan

Lecture No. 11

Software Design

6.1 Introduction

Recalling our discussion of software construction process, once the requirements of a software system have been established, we proceed to design that system. During the design phase, the focus shifts from what to how. That is, at this stage we try to answer the question of how to build the system. The objective of the design process is to analyze and understand the system in detail so that features and constituent components of at least one feasible solution are identified and documented. The design activity provides a roadmap to progressively transform the requirements through a number on stages into the final product by describing the structure of the system to be implemented.

It includes modeling of the data structures and entities, the physical and logical partitioning of the system into components, and the interfaces between different components of the system as well as interfaces to the outside world. Sometimes design of algorithms is also included in this activity.

6.2 Managing Complexity of a Software System

A complex system that works is invariably found to have evolved from a simple system that worked. The structure of a system also plays a very important role. It is likely that we understand only those systems that have hierarchical structure and where intra-component linkages are generally stronger than inter component linkages. To manage the complexity of the system we need to apply the principles of separation of concern, modularity, and abstraction. This leads to designs that are easy to understand and hence easy to maintain.

Separation of concern, modularity, and abstraction are different but related principles.

Separation of concern allows us to deal with different individual aspects of a problem by considering these aspects in isolation and independent of each other.

A complex system may be divided into smaller pieces of lesser complexity called modules. This is the classic divide-and-conquer philosophy – if you cannot solve a complex problem, try to break it into smaller problems that you can solve separately and then integrate them together in a systematic fashion to solve the original problem. One major advantage of modularity is that it allows the designer to apply the principle of separation of concern on individual modules.

© Copyright Virtual University of Pakistan

Software Design Process

Software design is not a sequential process. Design of a software system evolves through a number of iterations. The design process usually involves developing a number of different models, looking at the system from different angles and describing the system at various levels of abstraction. Like the various different models used during requirement engineering domain models, these models complement each other. As stated earlier, software design provides a road map for implementation by clearly describing how the software system is to be realized.

A activities performed at this stage include design of the software architecture by showing the division of system into sub-systems or modules, the specification of the services provided by these sub-systems and their interfaces with each other, division of each sub-system into smaller components and services and interfaces provided by each one of these components. Data modeling is also an essential activity performed during the design phase. This includes the identification of data entities and their attributes, relationships among these entities, and the appropriate data structures for managing this data.

Software Design Strategies

Software design process revolves around decomposing of the system into smaller and simpler units and then systematically integrates these units to achieve the desired results. Two fundamental strategies have been used to that end. These are functional or structured design and object oriented design.

In the functional design, the structure of the system revolves around functions. The entire system is abstracted as a function that provides the desired functionality (for example, the main function of a C program). This main function is decomposed into smaller functions and it delegates its responsibilities to these smaller functions and makes calls to these functions to attain the desired goal. Each of these smaller functions is decomposed into even smaller functions if needed. The process continues till the functions are defined at a level of granularity where these functions can be implemented easily. In this design approach, the system state, that is the data maintained by the system, is centralized and is shared by these functions.

The object-oriented design takes a different approach. In this case the system is decomposed into a set of objects that cooperate and coordinate with each other to implement the desired functionality. In this case the system state is decentralized and each object is held responsible for maintaining its own state. That is, the responsibility of marinating the system state is distributed and this responsibility is delegated to individual objects. The communication and coordination among objects is achieved through message passing where one object requests the other object if it needs any services from that object.

© Copyright Virtual University of Pakistan

The object-oriented approach has gained popularity over the structured design approach during the last decade or so because, in general, it yields a design that is more maintainable than the design produced by the functional approach.

Software Design Qualities

A software design can be looked at from different angles and different parameters can be used to measure and analyze its quality. These parameters include efficiency, compactness, reusability, and maintainability. A good design from one angle may not seem to be suitable when looked from a different perspective. For example, a design that yields efficient and compact code may not be very easy to maintain. In order to establish whether a particular design is good or not, we therefore have to look at the project and application requirements. For example, if we need to design an embedded system for the control of a nuclear reactor or a cruise missile, we would probably require a system that is very efficient and maintainability would be of secondary concern. On the other hand, in the case of an ordinary business system, we would have a reversal in priorities.

Maintainable Design

Since, in general, maintenance contributes towards a major share of the overall software cost, the objective of the design activity, in most cases, is to produce a system that is easy to maintain. A maintainable design is the one in which cost of system change is minimal and is flexible enough so that it can be easily adapted to modify exiting functionality and add new functionality.

In order to make a design that is maintainable, it should be understandable and the changes should be local in effect. That is, it should be such that a change in some part of the system should not affect other parts of the system. This is achieved by applying the principles of modularity, abstraction, and separation of concern. If applied properly, these principles yield a design that is said to be more cohesive and loosely coupled and thus is easy to maintain.

© Copyright Virtual University of Pakistan

Lecture No. 12

6.3 Coupling and Cohesion

Coupling is a measure of independence of a module or component. Loose coupling means that different system components have loose or less reliance upon each other. Hence, changes in one component would have a limited affect on other components.

Strong cohesion implies that all parts of a component should have a close logical relationship with each other. That means, in the case some kind of change is required in the software, all the related pieces are found at one place. Hence, once again, the scope is limited to that component itself.

A component should implement a single concept or a single logical entity. All the parts of a component should be related to each other and should be necessary for implementing that component. If a component includes parts that are not related to its functionality, then the component is said to have low cohesion.

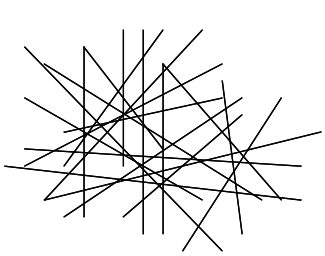
Coupling and cohesion are contrasting concepts but are indirectly related to each other. Cohesion is an internal property of a module whereas coupling is its relationship with other modules. Cohesion describes the intra-component linkages while couple shows the inter-component linkages. Coupling measures the interdependence of two modules while cohesion measures the independence of a module. If modules are more independent, they will be less dependent upon others. Therefore, a highly cohesive system also implies less coupling.

A good example of a system with a very high cohesion and very less (almost nil) coupling is the electric subsystem of a house that is made up of electrical appliances and wires. Since each one of the appliances has a clearly definable function that is completely encapsulated within the appliance. That means that an appliance does not depend upon any other appliance for its function. Therefore, each appliance is a highly cohesive unit. Since there are no linkages between different appliances, they are not coupled. Let us now assume that we have added a new centralized control unit in the system to control different appliances such as lights, air conditioning, and heating, according to certain settings. Since this control unit is dependent upon the appliances, the overall system has more coupling than the first one.

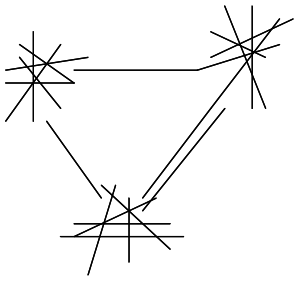
Modules with high cohesion and low coupling can be treated and analyzed as black boxes. This approach therefore allows us to analyze these boxes independent of other modules by applying the principle of separation of concern.

© Copyright Virtual University of Pakistan

Coupling and cohesion can be represented graphically as follows.



High Coupling



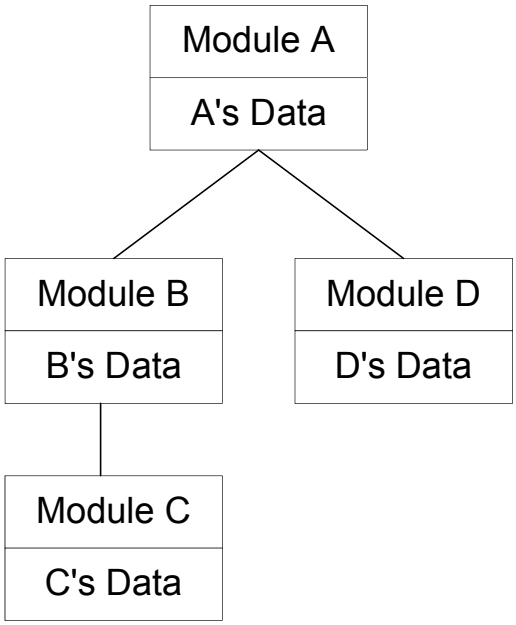
Low Coupling

This diagram depicts two systems, one with high coupling and the other one with low coupling. The lines depict linkages between different components. In the case of highly coupled system, module boundaries are not well defined, as everything seems to be connected with everything else. On the other hand, in the system with low coupling modules can be identified easily. In this case intra component linkages are stronger while inter component linkages are weak.

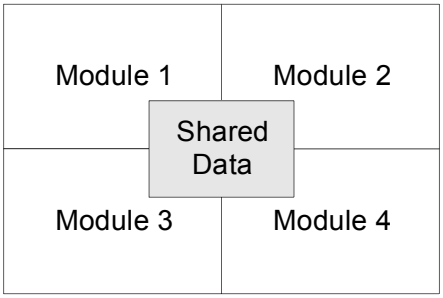
© Copyright Virtual University of Pakistan

Example of Coupling

The modules that interact with each other through message passing have low coupling while those who interact with each other through variables that maintain information about the state have high coupling. The following diagram shows examples of two such systems.



Low Coupling



High Coupling

In order to understand this concept, let us consider the following example. In this example, we have a class vector in which the data members have been put in the public part.

```
class vector {
  public:
        float x;
        float y;
        vector (float x, float y);
        float getX();
        float getY();
        float getMagnitude();
        float getAngle();
};
```

Now let us assume that we want to write a function to calculate dot product of two vectors. We write the following function.

© Copyright Virtual University of Pakistan

```
float myDotProduct1(vector a, vector b)
{
        float temp1 = a.getX() * b.getX();
        float temp2 = a.getY() * b.getY();
        return temp1 + temp2;
}
```

Since the data members are public, one could be enticed to use these members directly (presumably saving some function calls overhead) and rewrite the same function as follows:

```
float myDotProduct2(vector a, vector b)
{
        float temp1 = a.x * b.x;
        float temp2 = a.y * b.y;
        return temp1 + temp2;
}
```

So far, there does not seem to be any issue. But the scenario changes as soon as there are changes in the class implementation. Now let us assume that for some reason the class designer changes the implementation and data structure and decides to stores the angle and magnitude instead of the x and y components of the vector. The new class looks like as follows:

```
class vector {
  public:
        float magnitude;
        float angle;
        vector (float x, float y);
        vector (float magnitude, float angle);
        float getX();
        float getY();
        float getMagnitude();
        float getAngle();
};
```

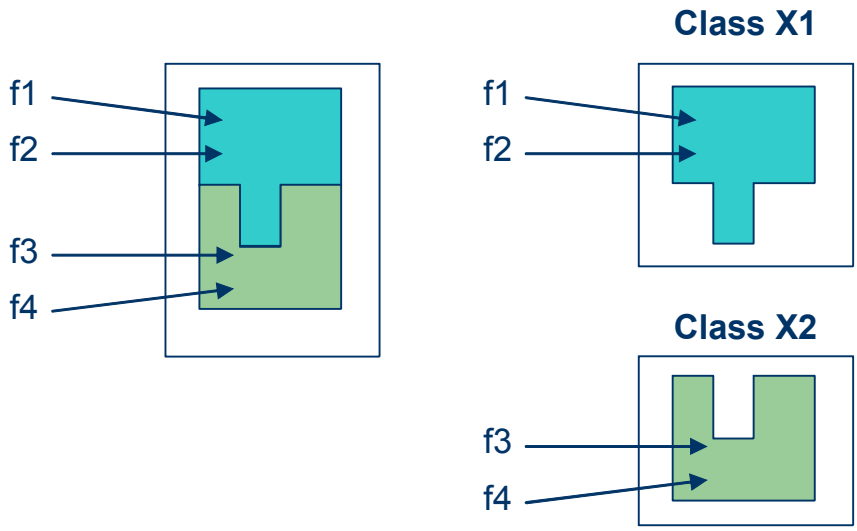
Now we see the difference in the two implementations of the dot product function written by the user of this class. In the first case, as the dot product function is dependent upon the public interface of the vector class, there will be no change while in the second case the function will have to be rewritten. This is because in the first case the system was loosely coupled while in the second case there was more dependency on the internal structure of the vector class and hence there was more coupling.

© Copyright Virtual University of Pakistan

Example of Cohesion

As mentioned earlier, strong cohesion implies that all parts of a component should have a close logical relationship with each other. That means, in case some kind of change is required in the software, all the related pieces are found at one place.

A class will be cohesive if most of the methods defined in a class use most of the data members most of the time. If we find different subsets of data within the same class being manipulated by separate groups of functions then the class is not cohesive and should be broken down as shown below.



© Copyright Virtual University of Pakistan

As an example, consider the following order class:

```
class order {
  public:
        int getOrderID();
        date getOrderDate();
        float getTotalPrice();
        int getCustometId();
        string getCustomerName();
        string getCustometAddress();
        int getCustometPhone();

        void setOrderID(int oId);
        void setOrderDate(date oDate);
        void setTotalPrice(float tPrice);
        void setCustometId(int cId);
        void setCustomerName(string cName);
        void setCustometAddress(string cAddress);
        void setCustometPhone(int cPhone);
        void setCustomerFax(int cFax)
  private:
        int oredrId;
        date orderDate;
        float totalPrice;
        item lineItems[20];
        int customerId;
        string customerName;
        int customerPhone;
        int customerFax;
};
```

The Order class shown above represents an Order entity that contains the attributes and behavior of a specific order. It is easy to see that this contains information about the order as well as the customer which is a distinct entity. Hence it is not a cohesive class and must be broken down into two separate classes as shown. In this case each on of these is a more cohesive class.

© Copyright Virtual University of Pakistan

```
class order {
   public:
        int getOrderID();
        date getOrderDate();
        float getTotalPrice();
        int getCustometId();

        void setOrderID(int oId);
        void setOrderDate(date oDate);
        void setTotalPrice(float tPrice);
        void setCustometId(int cId);
        void addLineItem(item anItem);
   private:
        int oredrId;
        date orderDate;
        float totalPrice;
        item lineItems[20];
        int customerId;
};

class customer {
   public:
        int getCustometId();
        string getCustomerName();
        string getCustometAddress();
        int getCustometPhone();
        int getCustomerFax();

        void setCustometId(int cId);
        void setCustomerName(string cName);
        svoid setCustometAddress(string cAddress);
        void setCustometPhone(int cPhone);
        void setCustomerFax(int cFax)
   private:
        int customerId;
        string customerName;
        int customerPhone;
        int customerFax;
};
```


© Copyright Virtual University of Pakistan

6.4 Abstraction and Encapsulation

Abstractions is a technique in which we construct a model of an entity based upon its essential characteristics and ignore the inessential details. The principle of abstraction also helps us in handling the inherent complexity of a system by allowing us to look at its important external characteristic, at the same time, hiding its inner complexity. Hiding the internal details is called encapsulation. In fact, abstraction is a special case of separation of concern. In this case we separate the concern of users of the entity who only need to understand its external interface without bothering about its actual implementation.

Engineers of all fields, including computer science, have been practicing abstraction for mastering complexity. Consider the following example.

```
void selectionSort(int a[], int size)
{
        int i, j, min, temp;
        for(i = 0; i < size –1; i++)
        {
                min = i;
                for (j = i; j < size; j++)
                {
                        if (a[j] < a[min])
                                min = j;
                }
                temp = a[i];
                a[i] = a[min];
                a[min] = temp;
        }
}
```

This function can be rewritten by abstracting out some of the logical steps into auxiliary functions. The new code is as follows.

```
viod swap(int &x, int &y)
{
        int temp;
        temp = x;
        x = y;
        y = temp;
}
```


© Copyright Virtual University of Pakistan

```
int indexOfMinimumValue(int a[], int from, int to)
{
        int i, min;
        min = from;
        for (i = from+1; i < to; i++)
                if (a[i] < a[min]) min = i;
        return min;
}

void selectionSort(int a[], int size)
{
        int i, min;
        for (i = 0; i < size; i++)
        {
                min = indexOfMinimumValue(a, i, size);
                swap(a[i], a[min]);
        }
}
```

In this function we have abstracted out two logical steps performed in this functions. These functions are finding the index of the minimum value in the given range in an array and swapping the minimum value with the value at the ith index in the array. It is easy to see that the resultant new function is easier to understand than the previous version of the selection sort function. In the process, as a by-product, we have created two auxiliary function mentioned above, which are general in nature and hence can be used elsewhere as well. Principle of abstraction thus generates reusable self-contained components.

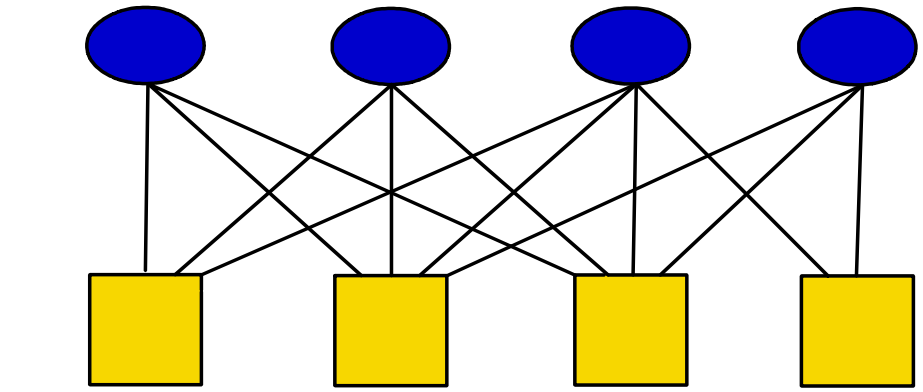
6.5 Function Oriented versus Object Oriented Design

Let us now try to understand the difference between object-oriented and function oriented (or action oriented) approach.

In the case of action-oriented approach, data is decomposed according to functionality requirements. That is, decomposition revolves around function. In the OO approach, decomposition of a problem revolves around data. Action-oriented paradigm focuses only on the functionality of a system and typically ignores the data until it is required. Object-oriented paradigm focuses both on the functionality and the data at the same time. The basic difference between these two is decentralized control mechanism versus centralized control mechanism respectively. Decentralization gives OO the ability to handle essential complexity better than action-oriented approach.

This difference is elaborated with the help of the following diagram:

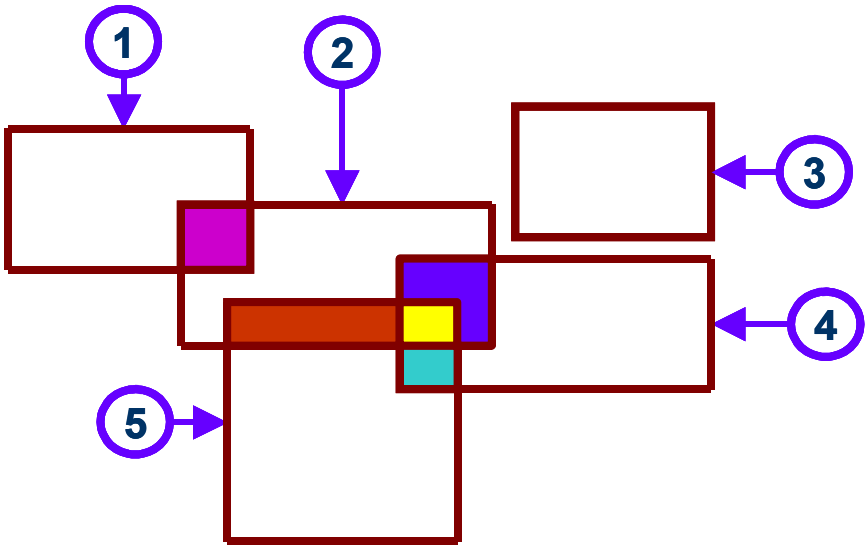
© Copyright Virtual University of Pakistan



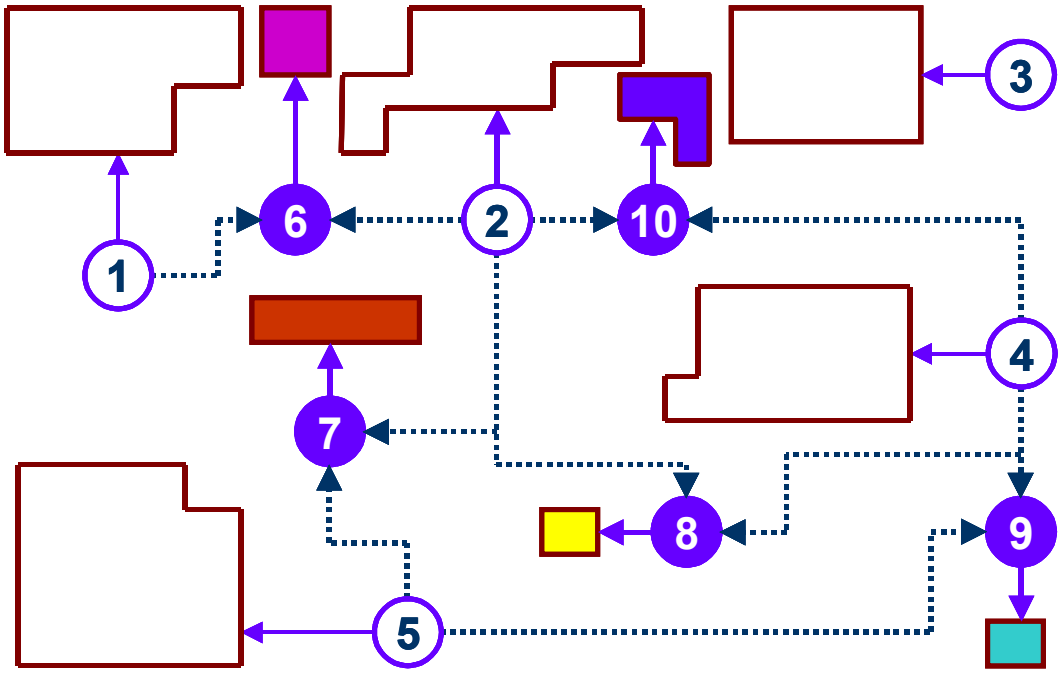
In this diagram, the ovals depict the function while rectangles/squares depict data. Since a function contains dynamic information while data contains only static information, if the function and data are managed separately, the required data components can be found by scanning a function but the functions that use a particular data cannot be found by just looking at the data. That is, the function knows about the data it needs to use but the data do not know about the functions using it. That means it is easy to make a change in a function since we would know which data components would be affected by this change. On the other hand, changing a data structure would be more difficult because it would not be easy to find all the functions that are using this data and hence also need to be modified.

Object oriented approach solves this problem by putting the relevant data and functionality together at one place. Hence, in case of a change, the effected components can be identified easily and the effect of change is localized. Therefore, maintenance becomes relatively easy as compared to function-oriented approach. This is made possible because the data is not shared in this case. Anyone needing any information contained in there would request the encapsulating object by sending it a message through the interface provided by the object. In this case we create highly cohesive objects by keeping the related data and function at one place and spinning-off non-related information into other classes. This can be elaborated with the help of the following diagram.

© Copyright Virtual University of Pakistan



Let us assume that the circles represent sets of functions and rectangles represent data that these function use to carry out their operation. In the object-oriented design, the data areas that are common among different sets of functions would be spun-off into their own classes and the user function would use these data through their interfaces only. This is shown in the following diagram.



© Copyright Virtual University of Pakistan

Lecture No. 13

Object Oriented Analysis and Design

Object Oriented Design - Why?

Software is primarily used to represent real-life players and processes inside a computer. In the past, software was considered as a collection of information and procedures to transform that information from input to the output format. There was no explicit relationship between the information and the processes which operate on that information. The mapping between software components and their corresponding real-life objects and processes was hidden in the implementation details. There was no mechanism for sharing information and procedures among the objects which have similar properties. There was a need for a technology which could bridge the gap between the real-life objects and their counter-parts in a computer. Object oriented technology evolved to bridge the gap. Object-oriented technology helps in software modeling of real-life objects in a direct and explicit fashion, by encapsulating data and processes related to a real-life object or process in a single software entity. It also provides a mechanism so that the object can inherit properties from their ancestors, just like real-life objects.

A complex system that works is invariably found to have evolved from a simple system that worked. The structure of a system also plays a very important role. It is likely that we understand only those systems which have hierarchical structure and where intra-component linkages are generally stronger than inter component linkages. That leads to loose coupling, high cohesion and ultimately more maintainability which are the basic design considerations. Instead of being a collection of loosely bound data structures and functions, an object-oriented software system consists of objects which are, generally, hierarchical, highly cohesive, and loosely coupled.

Some of the key advantages which make the object-oriented technology significantly attractive than other technologies include:

- Clarity and understandability of the system, as object-oriented approach is closer to the working of human cognition.
- Reusability of code resulting from low inter-dependence among objects, and provision of generalization and specialization through inheritance.
- Reduced effort in maintenance and enhancement, resulting from inheritance, encapsulation, low coupling, and high cohesion.

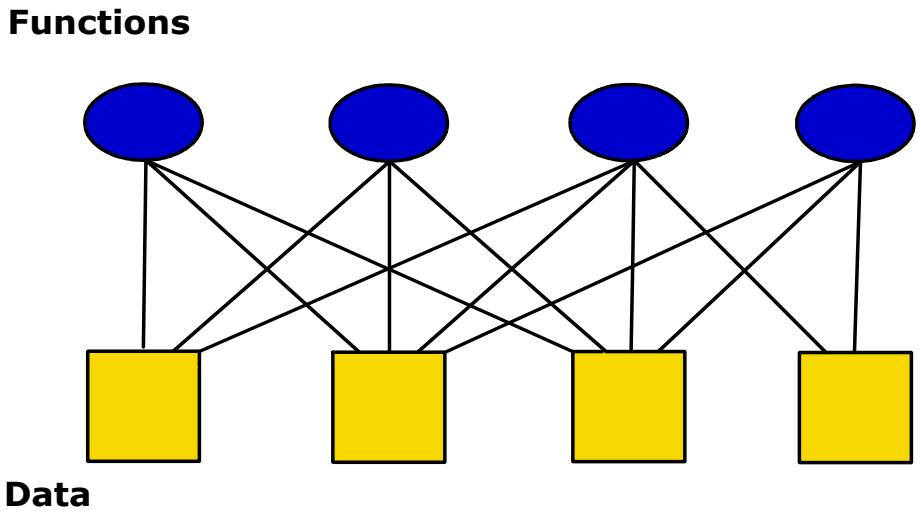
Difference between object-oriented and function-oriented design

Before talking about how to derive and object-oriented design, we first need to understand the basic difference between object-oriented and function oriented (or action oriented) approach.

© Copyright Virtual University of Pakistan

In the case of action-oriented approach, data is decomposed according to functionality requirements. That is, decomposition revolves around function. In the OO approach, decomposition of a problem revolves around data. Action-oriented paradigm focuses only on the functionality of a system and typically ignores the data until it is required. Object-oriented paradigm focuses both on the functionality and the data at the same time. The basic difference between these two is decentralized control mechanism versus centralized control mechanism respectively. Decentralization gives OO the ability to handle essential complexity better than action-oriented approach.

This difference is elaborated with the help of the following diagram:



In this diagram, the ovals depict the function while rectangles/squares depict data. Since a function contains dynamic information while data contains only static information, if the function and data are managed separately, the required data components can be found by scanning a function but the functions that use a particular data cannot be found by just looking at the data. That is, the function knows about the data it needs to use but the data do not know about the functions using it. That means, it is easy to make a change in a function since we would know which data components would be affected by this change. On the other hand, changing a data structure would be difficult because it would not be easy to find all the functions that are using this data and hence also need to be modified.

In the case of OO design since data and function are put together in one class, hence, in case of a change, the effected components can be identified easily and the effect of change is localized. Therefore, maintenance becomes relatively easy as compared to function-oriented approach.

© Copyright Virtual University of Pakistan

Object Oriented Design Components - What?

The Object and the Class

The basic unit of object oriented design is an object. An object can be defined as a tangible entity that exhibits some well defined behavior. An object represents an individual, identifiable item, unit, or entity, either real or abstract, with a well defined role in the problem domain. An object has state, behavior, and identity.

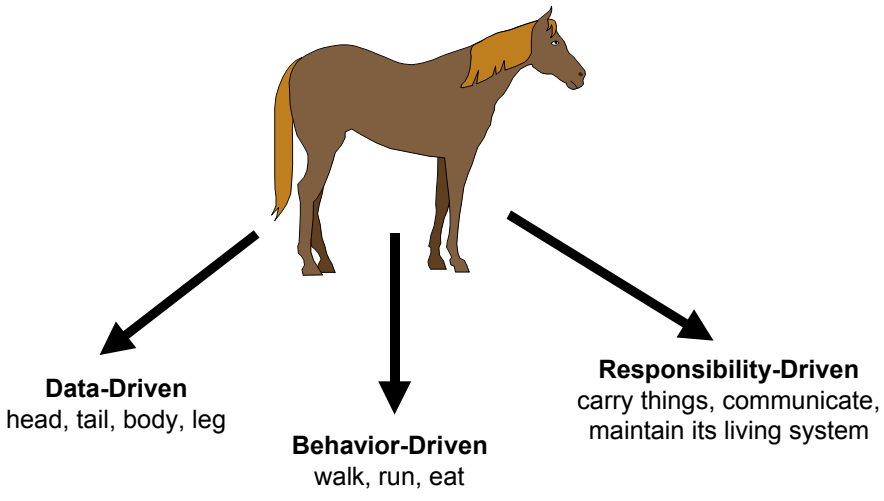
The state of an object encompasses all of the properties of the object and their current values. A property is an inherent or distinctive characteristic. Properties are usually static. All properties have some value. The state of an object is encapsulated within the object.

Behavior is how an object acts and reacts in terms of its state changes and message passing. The behavior of an object is completely defined by its actions. A message is some action that one object performs upon another in order to elicit a reaction. The operations that clients may perform upon an object are called methods.

The structure and behavior of similar objects are defined in their common class. A class represents an abstraction - the essence or the template of an object. A class specifies an interface (the outside view - the public part) and defines an implementation (the inside view - the private part). The interface primarily consists of the declaration of all the operations applicable to instances of this class. The implementation of a class primarily consists of the implementation of all the operations defined in the interface of the class

Classification

The most important and critical stage in the OOA and OOD is the appropriate classification of objects into groups and classes. Proper classification requires looking at the problem from different angles and with an open mind. When looked at from different perspectives and analyzed with different set of characteristics, same object can be classified into different categories. Let us try to understand this with the help of an example.



© Copyright Virtual University of Pakistan

Here, we can take a data-driven, behaviour driven, or responsibility driven perspective and will categorize the horse accordingly.

The Object Model

The elements of object oriented design collectively are called the Object Model. The object model encompasses the principles of abstraction, encapsulation, and hierarchy or inheritance.

Abstraction is an extremely powerful technique for dealing with complexity. Unable to master the entirety of a complex object, we ignore its essential details, dealing instead with generalized, idealized model of the object. An abstraction focuses on the outside view of an object, and hence serves to separate an objects external behavior from its implementation. Deciding upon the right set of abstractions for a given domain is the central problem in object oriented design.

Abstraction and encapsulation are complementary concepts. Abstraction provides the outside view to the client and encapsulation prevents clients from seeing its inside view. For abstraction to work, implementation must be encapsulated. Encapsulation hides the details of the implementation of an object. Intelligent encapsulation localizes design decisions that are likely to change. The ability to change the representation of an object without disturbing any of its clients is the essential benefit of encapsulation.

Relationship Among Objects

The object model presents a static view of the system and illustrates how different objects collaborate with one another through patterns of interaction. Inheritance, association and aggregation are the three inter-object relationships specified by the object model.

Inheritance defines a "kind of" hierarchy among classes. By inheritance, we specify generalization/specialization relationship among objects. In this relationship, a class (called the subclass) shares the structure and behavior defined in another class (called the superclass). A subclass augments or redefines the existing structure and behavior of its superclass. By classifying objects into groups of related abstractions, we come to explicitly distinguish the common and distinct properties of different objects, which further help us to master their inherent complexity. Identifying the hierarchy within a complex system requires the discovery of patterns among many objects.

In an association relationship, when object A "uses" object B, then A may send messages to B. The relationship defines visibility among objects.

The aggregation relationship defines part-of structure among objects. When object A is part of the state of object B, A is said to be contained by B. There are some tradeoffs between aggregation and association relationships. Aggregation reduces the number of objects that must be visible at the level of enclosing objects and may lead to undesirable tighter coupling among objects.

© Copyright Virtual University of Pakistan

Aggregation and Association - Conceptual and Implementation Issues and Differences

Association and Aggregation - Some basic differences

Objects do not exist in isolation. They rather collaborate with one another in many different ways to achieve an overall goal. The different types of relationships in which these objects are involved include association, aggregation, and inheritance. Briefly, inheritance denotes a "kind of" relationship, aggregation denotes a "part of" relationship, and association denotes some semantic connection among otherwise unrelated classes. Any further elaboration on inheritance relationship is beyond the scope of this discussion and therefore we shall concentrate on agrregation and association relationships only.

As mentioned earlier, aggregation is the "part-whole" or "a-part-of" relationship in which objects representing the components of something are encapsulated within an object representing the entire assembly. In other words, the whole is meaningless without its parts and the part cannot exist without its container or assembly. Some properties of the assembly propagate to the components as well, possibly with some local modifications. Unless there are common properties of components that can be attached to the assembly as a whole, there is little point in using aggregation. Therefore, as compared to association, aggregation implies a tighter coupling between the two objects which are involved in this relationship. Therefore, one way to differentiate between aggregation and association is that if the two objects are tightly coupled, that is, if they cannot exist independently, it is an aggregation, and if they are usually considered as independent, it is an association.

Object Creation and Life Time

From the object creation and life time point of view, when an object is instantiated, all of its parts must also be instantiated at the same time before any useful work can be done and all of its part die with it. While in the case of association, the life time of two associated object is independent of one another. The only limitation is that an object must be alive or has to be instantiated before a message can be sent to it.

Coupling and Linkages

As mentioned earlier, aggregation implies a much tighter coupling than association. In case of aggregation, the links between the whole and its part are permanent while in case of association the links may be maintained only just for the period an object requires the services of its associated object and may be disconnected afterwards.

Ownership and visibility

Another way of differentiating among the two is to look at them from the ownership and sharing point of view. In case of aggregation, since the whole contains the part, the part is encapsulated or hidden within the whole and is not accessible from outside while in case of association, the associated object may be used directly by other objects also. That is, in case of aggregation, only the whole is supposed to send a message to its parts while in case of association, anyone who holds a reference to it can communicate with it directly.

© Copyright Virtual University of Pakistan

In other words, in case of aggregation, the whole owns its parts and the part becomes a private property of the whole. For all practical purposes, any other object does not even need to know about its existence. On the other hand, an associated object may be shared among many different objects. That is, many different object may hold reference to the same object simultaneously.

Database persistence

From a database perspective, when an object is persisted or stored in the database, all of its components (all parts of the whole) must also be persisted in their entirety along with the "whole" for future reference while only a reference to the associated object may be stored in the database. Note that a normalized database would also enforce the above restriction.

© Copyright Virtual University of Pakistan

Lecture No. 14

Object Oriented Analysis

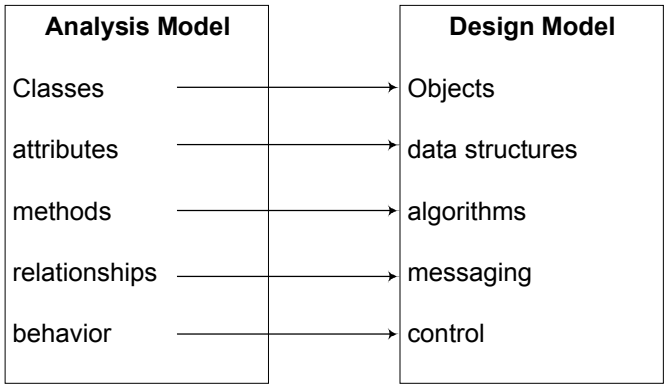
The intent of OOA is to define all classes, their relationships, and their behavior. A number of tasks must occur:

1) Static Model
   a) Identify classes (i.e. attributes and methods are defined)
   b) Specify class hierarchy
   c) Identify object-to-object relationships
   d) Model the object behavior
2) Dynamic Model
   a) Scenario Diagrams

Object Oriented Design

OOD transforms the analysis model into design model that serves as a blueprint for software construction. OOD results in a design that achieves a number of different levels of modularity. The four layers of the OO design pyramid are:

1) **The subsystem layer.** Contains a representation of each of the subsystems that enable the software to achieve its customers defined requirements and to implement the technical infrastructure that supports customer requirements.
2) **The class and object layer.** Contains the class hierarchies that enable the system to be created using generalization and increasingly more targeted specializations. The layer also contains design representations for each object.
3) **The message layer.** Contains the details that enable each object to communicate with its collaborators. This layer establishes the external and internal interfaces for the system.
4) **The responsibility layer.** Contains the data structures and algorithmic design for all attributes and operations for each object.



Translating the analysis model into a design model during object design

© Copyright Virtual University of Pakistan

Object-Oriented Analysis using Abbot's Textual Analysis

The first object-orientation technique that we will study is one of the oldest techniques to identify objects and their relationships. This technique is called Textual Analysis. It was initially developed by Abbot and then extended by Graham and others. In this technique different parts of speech are identified within the text of the specification and these parts are modeled using different components. The following table shows this scheme.

Part of speech	*Model component*	*Example*
proper noun	instance	Mehdi Hassan
improper noun	class/type/role	student, teacher
doing verb	operation	buy
being verb	classification	is a horse, is a book
having verb	composition	fan has wings
adjective	attribute value or class	this ball is green
adjective phrase	association	the customer with children
	operation	the customer who bought the kite

Once all the model components have been identified, we will eliminate the redundant or irrelevant components by again analyzing the text and the context of the problem.
Let's now try to understand this with the help of an example:

Problem Statement:
A simple cash register has a display, an electronic wire with a plug, and a numeric keypad, which has keys for subtotal, tax, and total. This cash storage device has a total key, which triggers the release on the drawer. The numeric buttons simply place a number on the display screen, the subtotal displays the current total, the tax key computes the tax, and the total key adds the subtotal to the tax.

Our task now is to:

- Identify all the classes in this problem statement.
- Eliminate the unnecessary classes.

We are now going to use nouns to find classes.

Nouns (initial)

Register	Display	Wire
Plug	Keypad	Keys
Devices	Release	Drawer
Buttons	Screen	Number

© Copyright Virtual University of Pakistan

Total Tax

Nouns (General Knowledge)

0-9 keys Money Subtotal Key
Tax Key Total Key

Eliminating Irrelevant/Redundant Nouns

We now analyze the identified nouns and try to establish whether they would be stand-alone classes in our domain or not. Outcome of this analysis is shown below.

Register
Display
Wire → Irrelevant
Plug → Irrelevant
Keypad
Keys
Devices → Vague
Release → Irrelevant
Drawer
Buttons → Redundant
Screen → Redundant
Number → Attribute
Total → Attribute
Tax → Attribute
0-9 Key
Value → Attribute
Money
Subtotal Key
Tax Key
Total Key

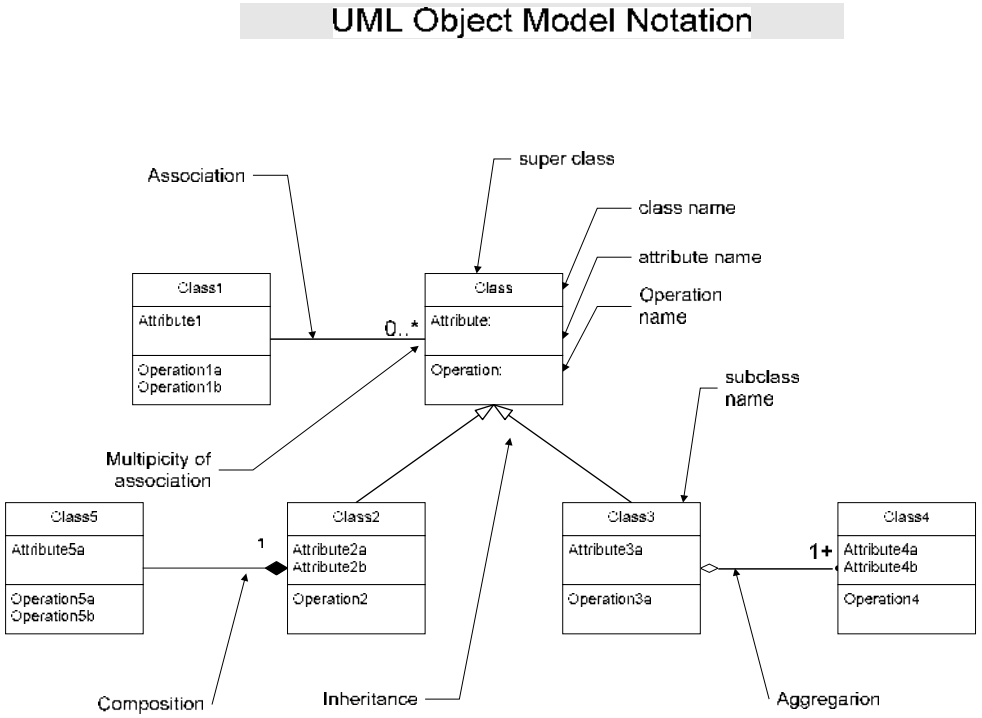
We will continue with technique to identify all the constituent components of the model and derive our object-oriented design.

© Copyright Virtual University of Pakistan

Lecture No. 15

The Notation

Many different notations are used for documenting the object oriented design. Most popular of these include, Rumbaugh, Booch, and Coad, and UML(Unified Modeling Language). We will be using UML to document our design. Although the notation is very comprehensive and detailed, but the key features of this notation are presented in the following diagram.



© Copyright Virtual University of Pakistan

Lecture No. 16

Derivation of the Object Model – The Coad Methodology

An object model of a system captures the static structure of a system by showing the objects in the systems, their relationships, their attributes, and their services. To stream line the derivation of the object model, Peter Coad has divided the process into 5 activities, each being further subdivided into a number of steps. Following is the description of these activities.

Select Objects – who am I?

We have used an approach that divides the objects into different categories to make it easier to find them and establish their attributes, services, and collaborations. This activity, consisting of 6 steps, can help you find objects and categorize them. These steps are:

Select actors

Actors are people and organizations that take part in the system under consideration. Examples of actors are: person, organization (agency, company, corporation, foundation). Note that we are talking about actors and not their "roles". e.g. a customer is a role that a person plays, so if we have a customer in our problem domain, we will also add a person as actor in the model.

Select Participants

A participant is a role that each actor plays in the system under consideration. Examples of participants are: agent, applicant, buyer, cashier, clerk, customer, dealer, distributor, donor, employee, investor, member, officer, owner, policy holder, recipient, student, supervisor, supplier, teacher, worker. It may be noted that the same person may play different roles at different times in the system. That means that if we model this behavior using Generalization-Specialization instead of Actor-Participant, we may end up with multiple inheritance.

Select Places

Places are where things come to rest or places that contain other objects. Examples of places are: airport, assembly-line, bank, city, clinic, country, depot, garage, hanger, hospital, plant, region, sales outlet, service center, shelf, station, store, warehouse, zone.

Select Transactions

Transactions are the "events" that must be remembered through time. These are entries that must be maintained in a historical record or log which may be used to answer questions or perform assessments. These transactions usually come from a window (GUI), some object which monitors for significant event and logs that information, or a another system that interacts with the system under consideration and logs some

© Copyright Virtual University of Pakistan

information. Examples of transactions are: agreement, assignment, authorization, contract, delivery, deposit, incident, inquiry, order, payment, problem report, purchase, refund, registration, rental, reservation, sale, shift, shipment, subscription, withdrawal. Note that nearly all transactions consist of a number of transaction line items.

Select Container Objects

Containers are objects that hold other objects. Note the similarity of definition between container and places. The difference is that a place is a place in the literal sense while a container is a any object that can hold other objects, e.g. bin, box, cabinet, folder, locker, safe, shelf, etc. Therefore a place is also a container but every container need not be a place.

Select Tangible things

Take a "walk" through the system and select "tangible" things around you used in the problem domain. These may be characterized as all the remaining (not yet selected) "nouns" that make up the problem domain. Examples are: account, book, calendar, cash box, cash drawer, item, plan, procedure, product, schedule, skill, tool, etc.

While selecting objects, the following considerations should be kept in mind for a simpler (and better) object model.

1. Every object that you put in your object model should have some responsibility or role to play in the problem domain. You need to know each object, its attributes, and services. If there is no way to know about the object, remove it from the object model.

2. Avoid having controller objects because controllers usually end up with functionality that's better done by other objects themselves, making the message passing more complicated, and resulting in higher coupling. Use delegation instead. Note the difference between controlling and delegation; a controller wants to do every thing by himself (doesn't trust anyone), while a good manager delegates responsibility (and takes credit).

3. In large systems several objects are likely to have similar or even identical responsibilities. Look for such objects and seek a common name to simplify the object model.

4. Use meaningful class names, names that describe objects in that class. Try to use names from the domain vocabulary to avoid confusion.

© Copyright Virtual University of Pakistan

Lecture No. 17

Identify Structures

A structure is a manner of organization which expresses a semantically strong organization within the problem domain. There are two type of structures: Generalization-Specialization (Gen-Spec) and whole-part. This activity covers the identification of these structures in the following 2 steps:

Identify Gen-Spec Structures (Hierarchy)

Consider each class that you have identified as a specialization and then look for its generalization and vice versa.

Identify Whole-Part structures (Aggregations) - What are my components?

For each object that you have identified, consider it as a whole and then try to find out its parts - objects that make up this object.

Define Attributes - What I Know?

The first two activities would identify most of the objects (classes) in the problem domain. Now is the time to think about the role and responsibilities of these objects. The first thing to consider is their attributes, i.e., what it knows.

For each object include the attributes that come to mind when you first think about the object. The criteria for the inclusion of an attribute is that it should be included if the system needs to know its value and it cannot get it any other way. Don not add an attribute for an association or aggregation. Examples of attributes are: number, name, address, date, time, operational state, phone, status, threshold, type, etc. In particular, consider the following attributes for different types of objects.

1. For actors consider name, address, phone.
2. For participants consider number, date and time, password, authorization level.
3. For place/location consider number, name, address (perhaps latitude, longitude, altitude).
4. For transaction consider number, date, time, status.
5. For line item consider quantity, status.
6. For item consider name, description, dimension, size, UPC, weight.

Like object selection, there are a number of issues that every designer must be aware of while defining attributes of an object. These are:

1. An attribute that varies over time, e.g., price of an item, should be replaced by an additional class with an effective date and value.
2. An attribute that may have a number of values should be replaced by a new class and an object connection.
3. Replace “yes/no” type attributes with “status” type attributes for flexibility.
4. If there are classes with common attributes and generalization-specialization makes good sense, then add a generalization class and factor out the commonality.

© Copyright Virtual University of Pakistan

Show Collaborations (associations and aggregations) - Who I know?

The second step in establishing each object's responsibility is to identify and show how this object collaborates with other objects, i.e., who it knows. These collaborations can be identified with the help of the following 8 steps:

1. For an actor, include an object connect to its participants (association).
2. For a participant, include an object connection to its actor (already established) and its transactions (association).
3. For a location, include object connections to objects that it can hold (association), to its part objects (aggregation), and to the transactions that are taking place at that location (association).
4. For transactions, include object connections to its participants (already established), its line items (aggregation), and its immediate subsequent transaction (aggregation).
5. For a transaction line item, include object connections to its transaction (already established), its item (association), a companion "item description" object (association), and a subsequent line item (association).
6. For an item, include object connections to transaction line item (already established), a companion "item description" object (association).
7. For a composite object, include object connections to its "part" object (aggregation).
8. For all objects (including all of the above) select connecting objects to which the object under consideration sends a message (within one or more scenarios) to get some information or to answer a query about objects directly related to it (association).

Define Services - What I do?

The third and last step in establishing each object's responsibility is to define what services does each object in the problem domain provide, i.e., what it does. Putting the right service with the right object is also very important since any mistake in judgment will increase coupling and reduce cohesion. The verbs in your problem domain usually indicate some of the services required of the associated object.

Software objects do things that the system is responsible to do with regard to that object. By putting the services with the attributes they work on results in lower coupling and stronger cohesion, and increased likelihood of reuse. The basic principle is to keep data and action together for lower coupling and better cohesion. The basic services, done by all (such as get, set, create, initialize), are not shown in the object model. While establishing the services for an object, the following fundamental questions should be asked:

1. Why does the system need this object any way?
2. What useful questions can it answer?
3. What useful action can it perform?
4. What this object can do, based upon what it knows?
5. What this object can do, based upon whom it knows?

© Copyright Virtual University of Pakistan

6. What calculations can it do?
7. What ongoing monitoring could it do?
8. What calculations across a collection could it make (letting each worker do its part)?
9. What selections across a collection could it make (letting each worker do its part)?

While establishing services of certain specific types of objects, the following should be considered:

1. For an actor, consider: calculate for me, rate me, is <value>, rank participants, calculate over participants.
2. For a participant, consider: calculate for me, rate me, is <value>, rank transactions, calculate over transactions.
3. For a place, consider: calculate for me, rate me, is <value>, rank transactions, calculate over contents, calculate over container line items.
4. For a Transaction, consider: calculate for me, rate me, is <value>, how many, how much, rank transaction line items, rank subsequent transactions, calculate over transaction line items, calculate over subsequent transactions.
5. For a line item, consider: calculate for me, rate me.
6. For an item, consider: calculate for me, rate me, is <value>, how many, how much, rank, calculate over specific items.

Lecture No. 18

CASE STUDY: Connie's Convenience Store - A point of Sale System

The System

Identify the purpose of the system

- develop an overall purpose statement in 25 words or less. Why this system? Why now?
- Keep the overall goal, the critical success factor, always before you.
- "To support, to help, to facilitate, ..."

Connie's Wish List

- scan items and automatically price them
- know whether an item is on sale
- automatically total the sale and calculate tax
- handle purcahses and returns
- handle payments with cash, check, or charge
- authorize checks and cards
- calculate change when working with cash or checks
- record all of the information about a customer transaction

© Copyright Virtual University of Pakistan

- balance the cash in the drawer with the amount recorded by the point-of-sale system.

Why ?

- speed up checkout time
- reduce the number of pricing errors
- reduce the labour required to ticket the item with a price, originally and when prices change.

Summary

to help each cashier work more effectively during checkout, to keep good records of each sale, and to store more efficient store operations.

Identify system features

Be certain to include features that cover the following

1. log important information
2. conduct business
3. analyze business results
4. interact with other systems

Identify features for logging important information

- to maintain prices based upon UPC
- to maintain tax categories (categories, rates, and effective dates)
- to maintain the authorized cashiers
- to maintain what items we sell in a store
- to log the results of each sale in a store

Identify features for conducting business

- to price each item, based upon its UPC
- to subtotal, calculate tax, and total
- to accept payment by cash, check, or charge

Identify features for analyzing business results

- to count how many of each item sold
- to count how much we received in cash, check, or credit card sales
- to assess how each cashier is performing
- to assess how each store is performing

Identify features for working with interacting systems

- to obtain authorization from one or more credit (or check) authorization system

© Copyright Virtual University of Pakistan

SELECTING OBJECTS

Select Actors

the actor is:

- person

Select Participants

the Participants are:

- cashier
- head cashier
- customer

Cashier and Head Cashier

Is there a difference between head cashier and cashier in terms of their behavior and knowledge?. If no then we don not need a separate class for head cashier.

Customer

customer. You must have a way to know about customer objects; otherwise it should not be put in the domain model.

Select Places

The places are:

- store
- shelf

Shelf

The system does not keep track of the shelves.

Select Transactions

Significant Transactions are:

- sale
- every sale is a collection of sale line items
- return
- payment
- session

Select Container Classes

The store is a container class.

a store contains

- cashiers
- registers
- items

Select Tangible Things

Tangible things in store:

© Copyright Virtual University of Pakistan

- item
- register
- cash drawer
- Tax Category (Descriptive things)

Session

Is it important? It is important in order to evaluate a cashier's performance.

Lecture No. 19

Identify Structures

Identify Gen-Spec Structures

Kinds of stores:

A store is a kind of sales outlet. Perhaps over time, Connie will expand to other kinds of sales outlets. Stores might be specialized into kinds of stores. For now on leave store as it is.

Kinds of sales:

- sales, returns

- only different is that the amount is positive or negative. Is there any other difference?

Prices:

- regular price, and promotional (sales) price

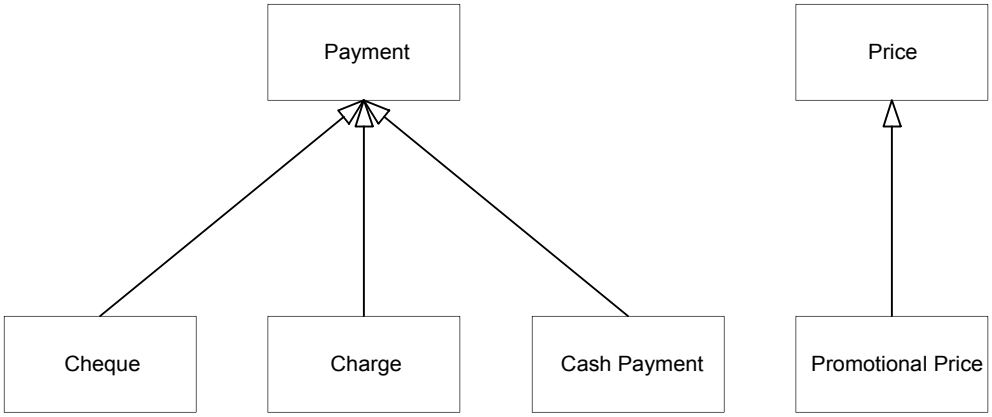
Payment:

- cash, check, and charge are kind of payments

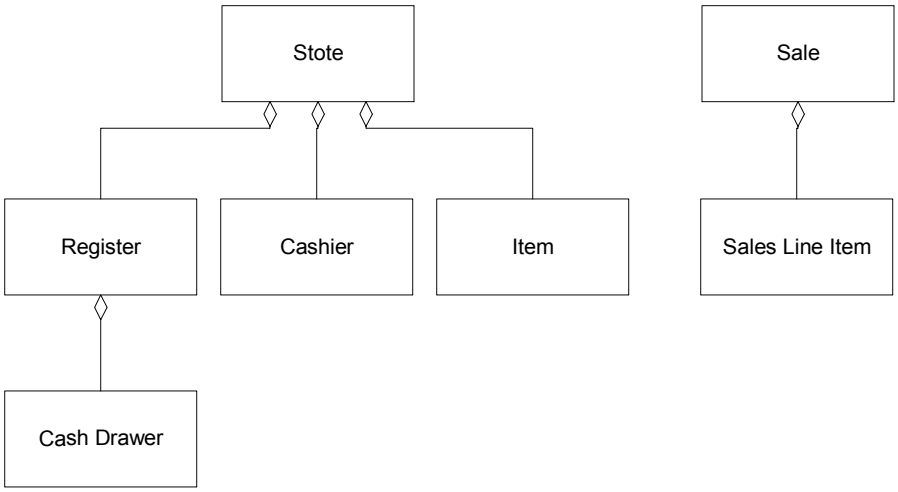
Identify Whole-Part Structures

- A store as a whole is made up of cashiers, registers, and items.
- A register contains a cash drawer.
- A sale is constituted of sale line items.

© Copyright Virtual University of Pakistan



Object Hierarchy



Whole-Part Structures

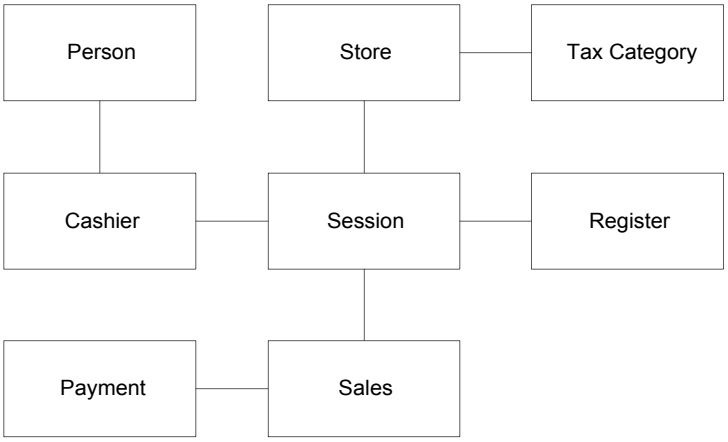
Establishing Responsibilities

Who I Know - Rules of Thumb

- an actor knows about its participants
  person knows about cashier
- a transaction knows about its participants
  a session knows about its register and cashier
- A transaction contains its transaction line items
  sale contains its sales line items

© Copyright Virtual University of Pakistan

- A transaction knows its sub transactions
  session knows about its sales
  sale knows about its payments
- A place knows about its transactions
  store knows about its sessions
- A place knows about its descriptive objects
  store knows about its tax categories
- A container knows about its contents
  a store knows about its cashiers, items, and registers



Association Relationships

Define Attributes, Services, and Links - What I know, What I do, and Who I know?

Actors:

person

Attributes: name, address, phone
Services: eating, walking

Participants:

cashier

Attributes: number, password, authorization level, current session
Services: isAuthorized, assess Performance

Places:

store

Attributes: name
Services: get item for UPC, get cashier for number

© Copyright Virtual University of Pakistan

Tangible things:

item

Attributes: number, description, UPCs, prices, taxable
attributes with repeating names - create new objects
UPC, Price (specialization - promotional price)
Services: get price for a date, how much for quantity
Who I Know? UPC, Price, tax category, sale line item

register

Attributes: number
Services: how much over interval, how many over interval
Who I know? store, session, cash drawer (part of register)

cash drawer

Attributes: balance, position (open, close), operational state
Services: open
Who I know? register

Tax Category

Attributes: category, rate, effective date
Services: just the basic services - get, add, set - don't show
Who I know? items?

Transactions:

sale

Attributes: date and time
Services: calculate subtotal, calculate total, calculate discount, calculate
tax, commit
Who I Know? session, payment, SLIs

sale line item

Attributes: date and time ?, quantity, tax status (regular, resale, tax-exempt)
Services: calculate sub total
Who I Know? item, sale

sale line item - how do you handle returns and sales

sale - you have control
return - more difficult
- return to a different store
- purchased for a different price
- returns an item no longer in the inventory

return

Attributes: return price, reason code, sale date, sale price

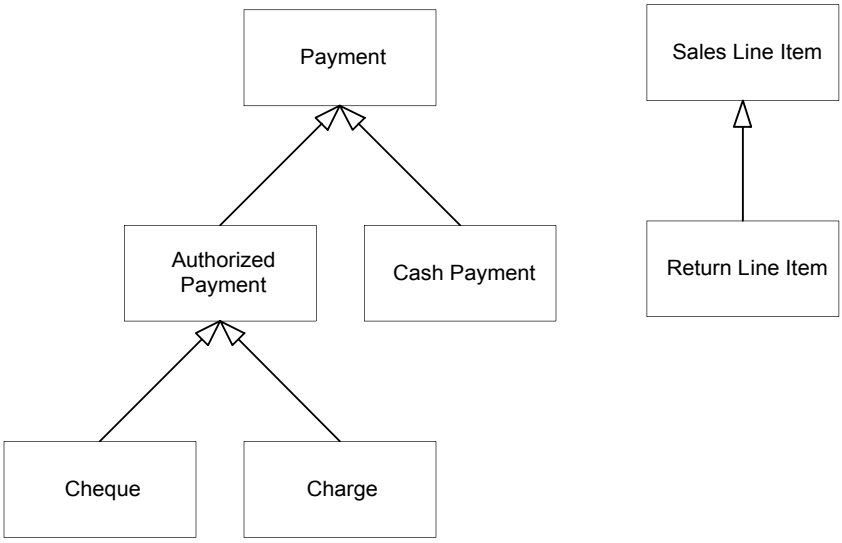
© Copyright Virtual University of Pakistan

Services:
Who I Know?

is it a case for gen-spec, what's same, what's different

payment - we have types of payments
Attributes:
each payment knows about its
amount paid, cash tendered
a check object knows its
bank, account number, amount tendered, authorization code
a credit object knows about its
card type, card number, expiration date, authorization code
common attributes among check and credit - use gen-spec
hierarchy becomes:
payment
cash payment
authorized payment
check
card

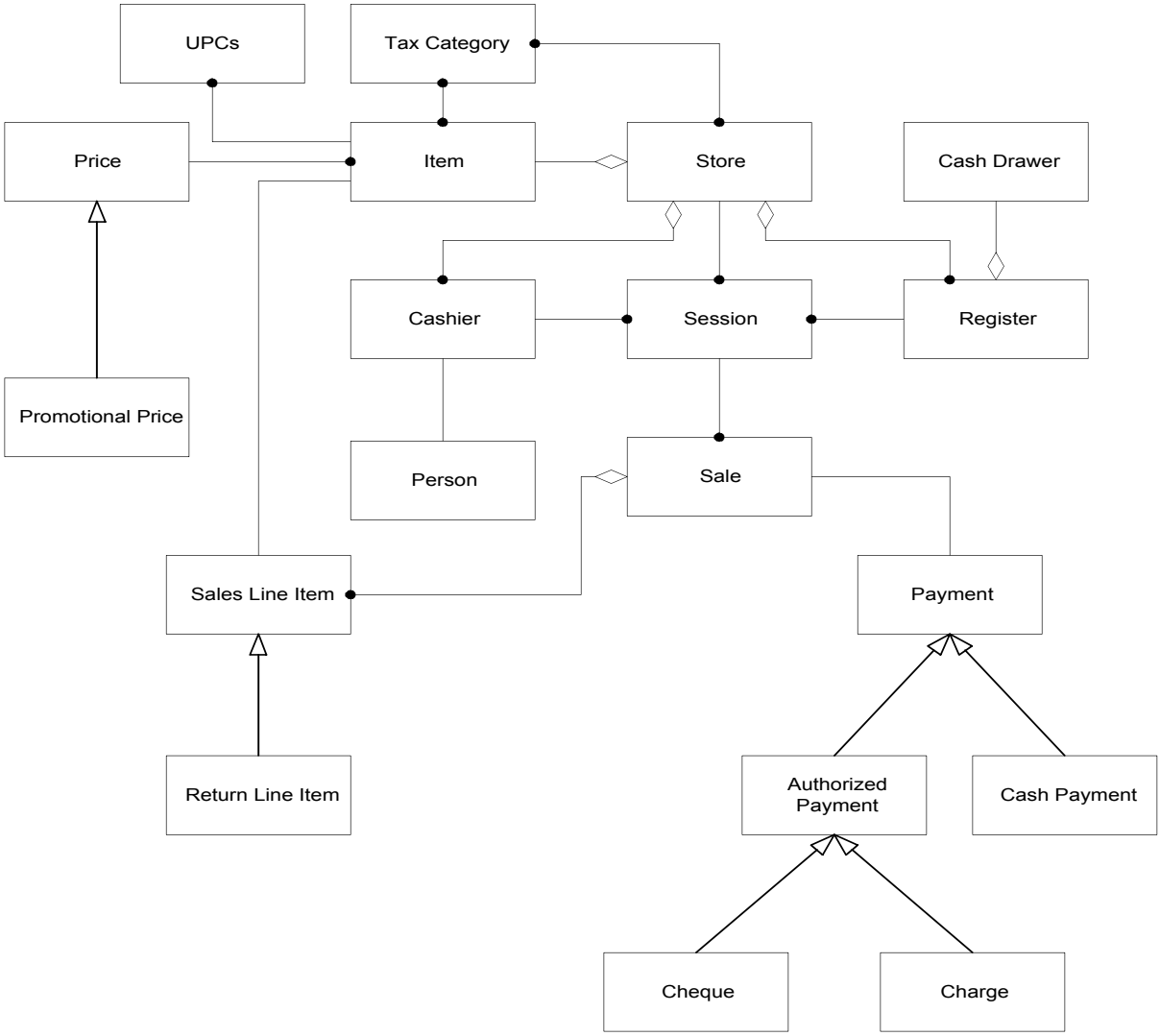
Services:
who I know: sale



session
Attributes: start date, end date, start time, end time
Services: how much money collected over interval, how many sales

© Copyright Virtual University of Pakistan

Who I know? register, cashier, store, sales



Object Model Diagram for Connie's Convenience Store

© Copyright Virtual University of Pakistan

Lecture No. 20

Interaction Diagrams – depicting the dynamic behaviour of the system

A series of diagrams can be used to describe the *dynamic behavior* of an object-oriented system. This is done in terms of a set of messages exchanged among a set of objects within a context to accomplish a purpose. This is often used to model the way a use case is realized through a sequence of messages between objects.

The purpose of Interaction diagrams is to:

- Model interactions between objects
- Assist in understanding how a system (a use case) actually works
- Verify that a use case description can be supported by the existing classes
- Identify responsibilities/operations and assign them to classes

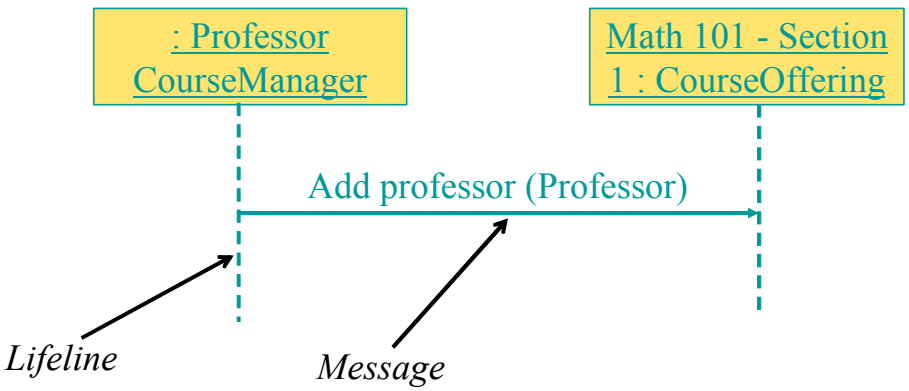
UML provides two different mechanisms to document the dynamic behaviour of the system. These are sequence diagrams which provide a time-based view and *Collaboration Diagrams* which provide an organization-based view of the system's dynamics.

The Sequence Diagram

Let us first look at Sequence Diagrams. These diagrams illustrate how objects interacts with each other and emphasize time ordering of messages by showing object interactions arranged in time sequence. These can be used to model simple sequential flow, branching, iteration, recursion and concurrency. The focus of sequence diagrams is on objects (and classes) and message exchanges among them to carry out the scenarios functionality. The objects are organized in a horizontal line and the events in a vertical time line.

The Notation

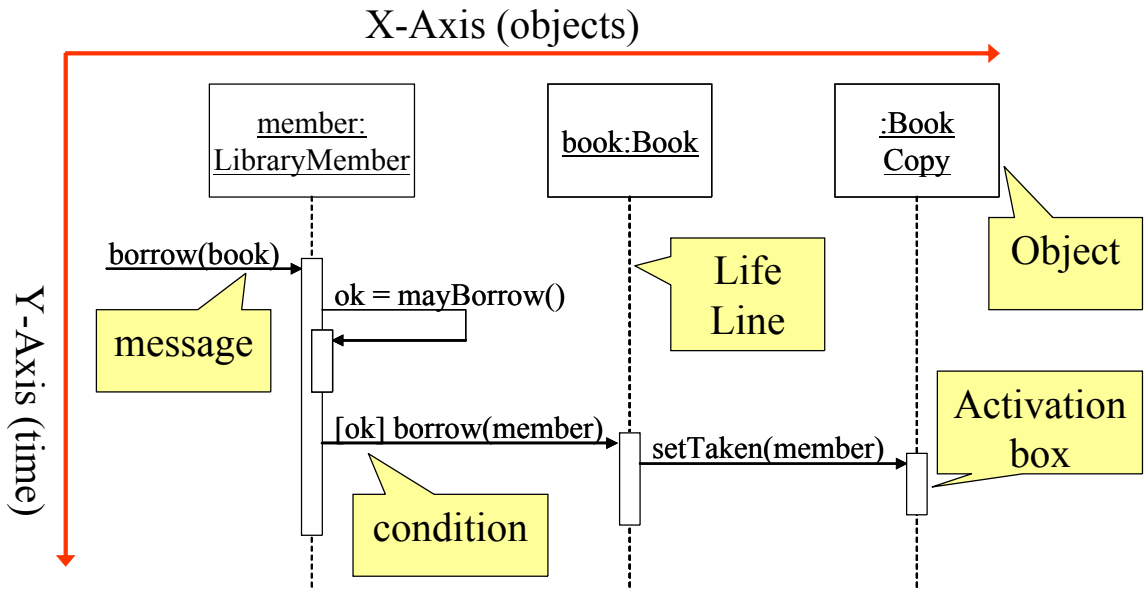
Following diagram illustrates the notation used for drawing sequence diagrams.



© Copyright Virtual University of Pakistan

The boxes denote objects (or classes), the solid lines depict messages being sent from one object to the other in the direction of the arrow, and the dotted lines are called life-lines of objects. The life line represents the object's life during interaction. We will discuss this in more detail later.

These concepts are further elaborated with the help of the following sequence diagram.



As shown above, in a sequence diagram, objects (and classes) are arranged on the X-Axis (horizontally) while time is shown on the Y-Axis (vertically). The boxes on the life-line are called activation boxes and show for how long a particular message will be active, from its start to finish. We can also show if a particular condition needs to occur before a message is invoked simply by putting the condition in a box before the message. For example, object *member:LibraryMember* sends a message to object *book:book* if the value of *ok* is true.

The syntax used for naming objects in a sequence diagram is as follows:

- syntax: [instanceName][:className]
- Name classes consistently with your class diagram (same classes).
- Include instance names when objects are referred to in messages or when several objects of the same type exist in the diagram.

An interaction between two objects is performed as a message sent from one object to another. It is most often implemented by a simple operation call. It can however be an

© Copyright Virtual University of Pakistan

actual message sent through some communication mechanism, either over the network or internally on a computer.

If object obj_1 sends a message to another object obj_2 an association must exist between those two objects. There has to be some kind of structural dependency. It can either be that obj_2 is in the global scope of obj_1, or obj_2 is in the local scope of obj_1 (method argument), or obj_1 and obj_2 are the same object.

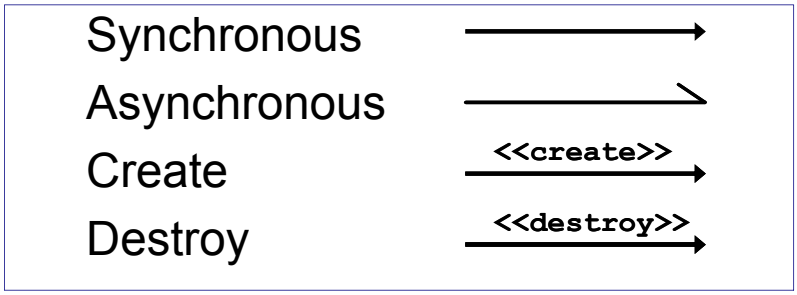
A message is represented by an arrow between the life lines of two objects. Self calls are also allowed. These are the messages that an object sends to itself. This notation allows self calls. In the above example, object *member:LibraryMember* sends itself the *mayBorrow* message. A message is labeled at minimum with the message name.
Arguments and control information (conditions, iteration) may also be included. It is preferred to use a brief textual description whenever an *actor* is the source or the target of a message.

The time required by the receiver object to process the message is denoted by an *activation-box.*

Lecture No. 21

Message Types

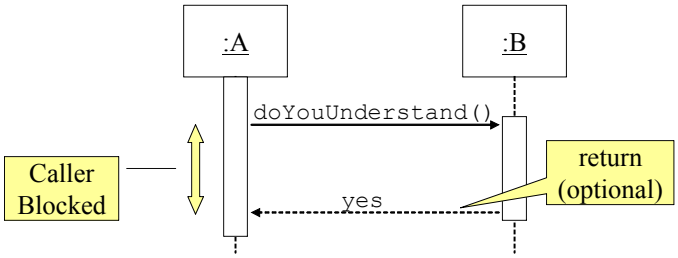
Sequence diagrams can depict many different types of messages. These are: synchronous or simple, asynchronous, create, and destroy. The following diagram shows the notation and types of arrows used for these different message types.



Synchronous Messages

Synchronous messages are "call events" and are denoted by the full arrow. They represent nested flow of control which is typically implemented as an operation call. In case of a synchronous message, the caller waits for the called routine to complete its operation before moving forward. That is, the routine that handles the message is completed before the caller resumes execution. Return values can also be optionally indicated using a dashed arrow with a label indicating the return value. This concept is illustrated with the help of the following diagram.

© Copyright Virtual University of Pakistan



While modeling synchronous messages, the following guidelines should be followed:

- Don't model a return value when it is obvious what is being returned, e.g. getTotal()
- Model a return value only when you need to refer to it elsewhere, e.g. as a parameter passed in another message.
- Prefer modeling return values as part of a method invocation, e.g.
  ok = isValid()

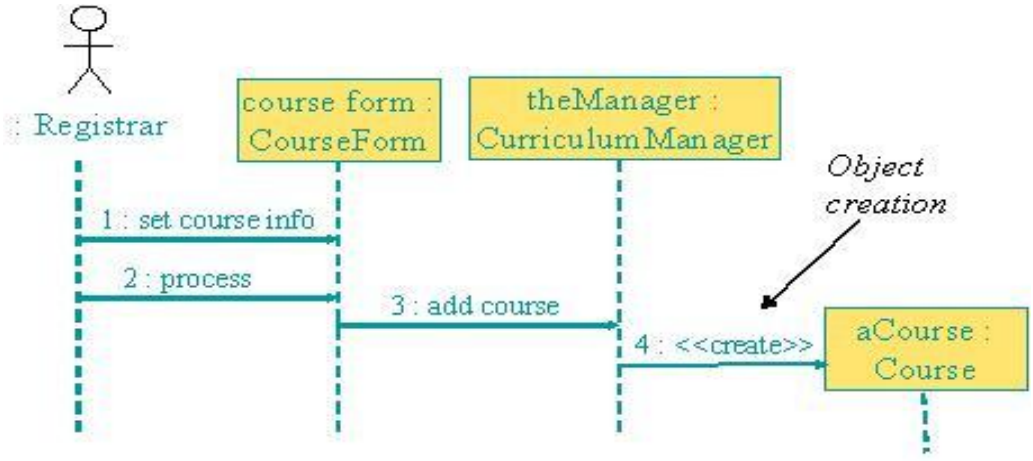
Asynchronous messages

Asynchronous messages are "signals," denoted by a half arrow. They do not block the caller. That is, the caller does not wait for the called routine to finish its operation for continuing its own sequence of activities. This occurs in multi-threaded or multi-processing applications where different execution threads may pass information to one another by sending asynchronous messages to each other. Asynchronous messages typically perform the following actions:

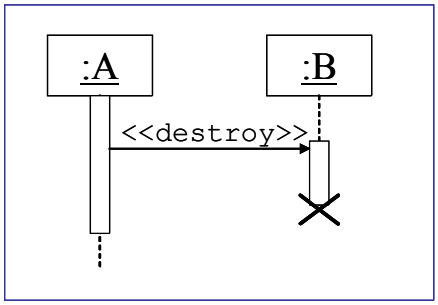
- Create a new thread
- Create a new object
- Communicate with a thread that is already running

Object Creation and Destruction

An object may create another object via a <<create>> message. Similarly an object may destroy another object via a <<destroy>> message. An object may also destroy itself. One should avoid modeling object destruction unless memory management is critical. The following diagrams show object creation and destruction. It is important to note the impact of these activities on respective life lines.



© Copyright Virtual University of Pakistan



Sequence diagrams and logical complexity

It is important to judiciously use the sequence diagrams where they actually add value. The golden principle is to keep it small and simple. It is important to understand that the diagrams are meant to make things clear. Therefore, in order to keep them simple, special attentions should be paid to the conditional logic. If it is simple then there is no harm in adding it to the diagram. On the other hand if the logic is complex then we should draw separate diagrams like flow charts.

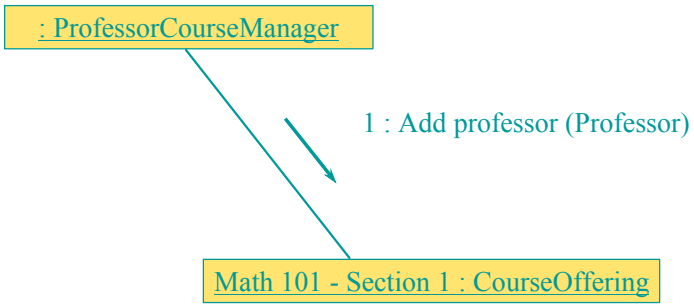
© Copyright Virtual University of Pakistan

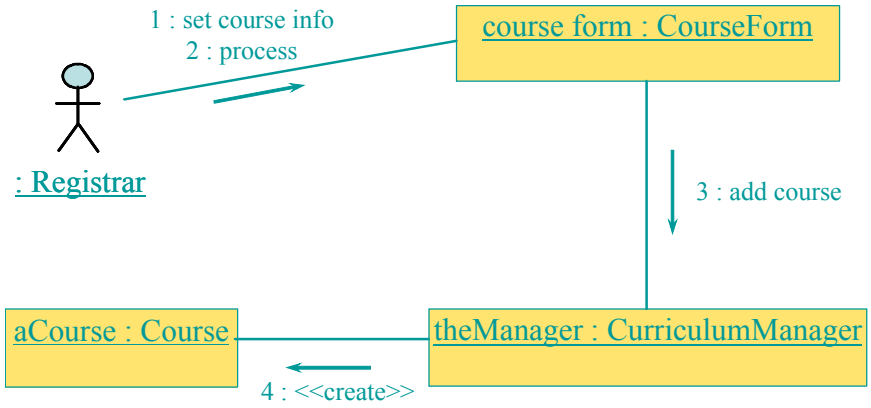
Collaboration diagrams

Collaboration diagrams can also be used to depict the dynamic behaviour of a system. They show how objects interact with respect to organizational units (boundaries!). Since a boundary shapes communication between system and outside world e.g. user interface or other system, collaboration diagrams can be used to show this aspect of the system. The sequence of messages determined by numbering such as 1, 2, 3, 4, … This shows which operation calls which other operation.

Collaboration diagrams have basically two types of components: objects and messages. Objects exchange messages among each-other. Collaboration diagrams can also show synchronous, asynchronous, create, and destroy message using the same notation as used in sequence diagrams. Messages are numbered and can have loops

The following diagrams illustrate the use of collaboration diagrams.





Comparing sequence & collaboration diagrams

© Copyright Virtual University of Pakistan

Sequence diagrams are best to see the flow of time. On the other hand, static object connections are best represented by collaboration diagrams. Sequence of messages is more difficult to understand in collaboration diagrams than in the case of sequence diagrams. On the other hand, object organization with control flow is best seen through collaboration diagrams. It may be noted that complex control is difficult to express anyway but collaboration diagrams can become very complex very quickly.

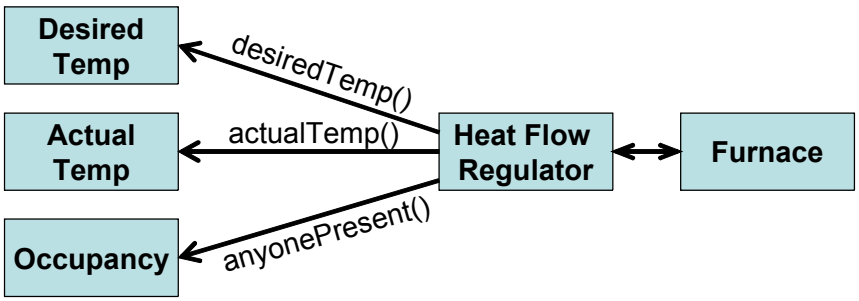
Evaluating the Quality of an Object-Oriented Design

Judging the quality of a design is difficult. We can however look at certain object-oriented design attributes to estimate its quality. The idea is to analyze the basic principle of encapsulation and delegation to judge whether the control is centralized or distributed, hence judging the coupling and cohesion in a design. This will tell us how maintainable a design is.

You may also recall our earlier discussion of coupling and cohesion. It can be easy to see that OO design yield more cohesive and loosely coupled systems.

The issue of centralized versus distributed control can be illustrated with the help of the following example.

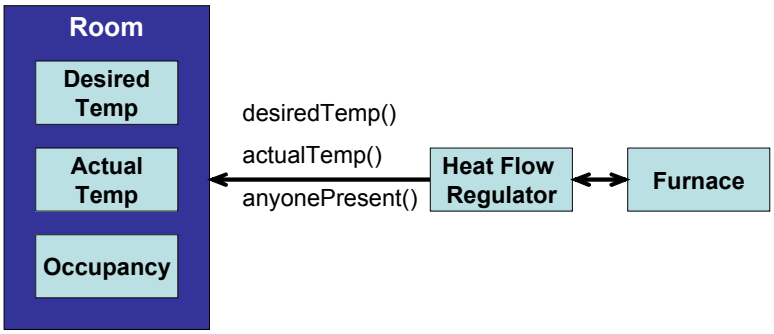
Consider a heat regulatory system for a room as shown below.



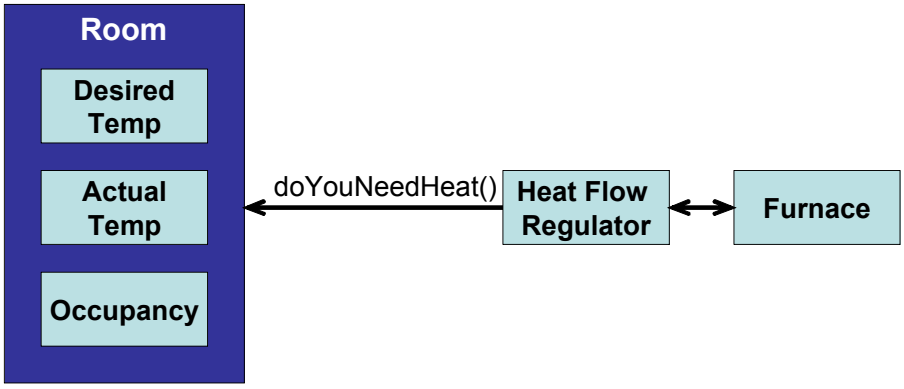
In this case, the room is not encapsulated as one entity and three different objects namely *Desired Temp, Actual Temp,* and *Occupancy* maintain necessary information about a room. In this case the Heat Flow Regulator has to communicate with three different objects.

© Copyright Virtual University of Pakistan

If we encapsulate the three objects into one Room object as shown below, then the Heat Flow Regulator will need to communicate with one object, hence the overall coupling of the system will be reduced.



This happened because in the first case intelligence is distributed while in the second case it is encapsulated. However, the control is still centralized as the Heat Flow Regulator has the control logic that first analyses the values from different queries and then makes a decision about turning the furnace on of off. We can improve the situation even further by delegating this responsibility to the Room object as shown below.



By doing that we reduce coupling even further because now we have made Room more cohesive by putting the function with the related data and have thus reduced the number and types of messages being sent from the regulator to the room.

That is, we can reduce the coupling of a system by minimizing the number of messages in the protocol of a class. The problem with large public interfaces is that you can never find what you are looking for…smaller public interfaces make a class easier to understand and modify. This can be further elaborated with the help of the following example. Suppose we have two functions defined for setting the desired temperature in the room:

© Copyright Virtual University of Pakistan

- SetMinimumValue(int aValue)
- SetMaximimumValue(int aValue)

We can reduce the total number of messages in the protocol of this class by consolidation these as shown below, hence reducing the overall complexity of the protocol.

- SetLimits(int minValue, int maxValue)

It is however important to use these kinds of heuristics judiciously and care must be taken so that the scope of the function does not go beyond providing one single operation.

© Copyright Virtual University of Pakistan

Lecture No. 22

Software and System Architecture

Introduction

When building a house, the architect, the general contractor, the electrician, the plumber, the interior designer, and the landscaper all have different views of the structure. Although these views are pictured differently, all are inherently related: together, they describe the building's architecture. The same is true with software architecture. Architectural design basically establishes the overall structure of a software system.

The design process for identifying the sub-systems making up a system and the framework for sub-system control and communication is *architectural design.* The output of this design process is a description of the *software architecture.*

The study of software architecture is in large part a study of software structure that began in 1968 when Edsger Dijkstra pointed out that it pays to be concerned with how software is partitioned and structured, as opposed to simply programming so as to produce a correct result. Dijkstra was writing about an operating system, and first put forth the notion of a layered structure, in which programs were grouped into layers, and programs in one layer could only communicate with programs in adjoining layers. Dijkstra pointed out the elegant conceptual integrity exhibited by such an organization, with the resulting gains in development and maintenance ease.

David Parnas pressed this line of observation with his contributions concerning information-hiding modules, software structures, and program families.

A program family is a set of programs (not all of which necessarily have been or will ever be constructed) for which it is profitable or useful to consider as a group. This avoids ambiguous concepts such as "similar functionality" that sometimes arise when describing domains. For example, software engineering environments and video games are not usually considered to be in the same domain, although they might be considered members of the same program family in a discussion about tools that help build graphical user interfaces, which both happen to use.[1]

Parnas argued that early design decisions should be ones that will most likely remain constant across members of the program family that one may reasonably expect to

© Copyright Virtual University of Pakistan

produce. In the context of this discussion, an early design decision is the adoption of a particular architecture. Late design decisions should represent trivially-changeable decisions, such as the values of compile-time or even load-time constants.

All of the work in the field of software architecture may be seen as evolving towards a paradigm of software development based on principles of architecture, and for exactly the same reasons given by Dijkstra and Parnas: Structure is important, and getting the structure right carries benefits.

Before talking about the software architecture in detail, let us first look at a few of its definitions.

8.2 What is Software Architecture?

What do we mean by software architecture? Unfortunately, there is yet no single universally accepted definition. Nor is there a shortage of proposed definition candidates. The term is interpreted and defined in many different ways. At the essence of all the discussion about software architecture, however, is a focus on reasoning about the *structural* issues of a system. And although architecture is sometimes used to mean a certain architectural style, such as client-server, and sometimes used to refer to a field of study, it is most often used to describe structural aspects of a particular system.

Before looking at the definitions for the software architecture, it is important to understand how a software system is defined. It is important because many definitions of software architecture make a reference to software systems.

According to UML 1.3, a system is a collection of connected units that are organized to accomplish a specific purpose. A system can be described by one or more models, possibly from different viewpoints. IEEE Std. 610.12-1990 defines a system as a collection of components organized to accomplish a specific function or set of functions. That is, a system is defined as an organized set of connected components to accomplish the specified tasks.

Let us now look at some of the software architecture definitions from some of the most influential writers and groups in the field.

Software Architecture Definitions

UML 1.3:

> Architecture is the organizational structure of a system. An architecture can be recursively decomposed into parts that interact through interfaces, relationships that connect parts, and constraints for assembling parts. Parts that interact through interfaces include classes, components and subsystems.

Bass, Clements, and Kazman. Software Architecture in Practice, Addison-Wesley 1997:

© Copyright Virtual University of Pakistan

'The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.

By "externally visible" properties, we are referring to those assumptions other components can make of a component, such as its provided services, performance characteristics, fault handling, shared resource usage, and so on. The intent of this definition is that a software architecture must abstract away some information from the system (otherwise there is no point looking at the architecture, we are simply viewing the entire system) and yet provide enough information to be a basis for analysis, decision making, and hence risk reduction."

Garlan and Perry, guest editorial to the *IEEE Transactions on Software Engineering*, April 1995:

Software architecture is "the structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time."

IEEE Glossary

Architectural design: The process of defining a collection of hardware and software components and their interfaces to establish the framework for the development of a computer system.

Shaw and Garlan

The architecture of a system defines that system in terms of computational components and interactions among those components. Components are such things as clients and servers, databases, filters, and layers in a hierarchical system. Interactions among components at this level of design can be simple and familiar, such as procedure call and shared variable access. But they can also be complex and semantically rich, such as client-server protocols, database accessing protocols, asynchronous event multicast, and piped streams.

Each of these definitions of software architecture, though seemingly different, emphasizes certain structural issues and corresponding ways to describe them. It is important to understand that although they are apparently different, they do not conflict with one another.

One can thus conclude from these definitions that an architectural design is an early stage of the system design process. It represents the link between specification and design processes. It provides an overall abstract view of the solution of a problem by identifying the critical issues and explicitly documenting the design choices made under the specified constraints as its primary goal is to define the non-functional requirements of a system and define the environment. It is often carried out in parallel with some specification

© Copyright Virtual University of Pakistan

activities and It includes the high-level design of modular components, their relationships and organization, and provides a foundation that one can build on to solve a problem.

8.3 Why is architecture important?

Barry Boehm says:

> *If a project has not achieved a system architecture, including its rationale, the project should not proceed to full-scale system development. Specifying the architecture as a deliverable enables its use throughout the development and maintenance process.*

Why is architecture important and why is it worthwhile to invest in the development of a architecture? Fundamentally, there are three reasons:

1. **Mutual communication.** Software architecture represents a common high-level abstraction of the system that most, if not all, of the system's stakeholders can use as a basis for creating mutual understanding, forming consensus, and communicating with each other.

   Each stakeholder of a software system (customer, user, project manager, coder, tester, and so on) is concerned with different characteristics of the system that are affected by its architecture. Architecture provides a common language in which different concerns can be expressed, negotiated, and resolved at a level that is intellectually manageable, even for large, complex systems. Without such language, it is difficult to understand large systems sufficiently to make the early decisions that influence both quality and usefulness.

2. **Early design decisions.** Software architecture represents the embodiment of the earliest set of design decisions about a system, and these early bindings carry weight far out of proportion to their individual gravity with respect to the system's remaining development, its service in deployment, and its maintenance life. It is also the earliest point at which the system to be built can be analyzed.

   An implementation exhibits an architecture if it conforms to the structural design decisions described by the architecture. This means that the implementation must be divided into the prescribed components, the components must interact with each other in the prescribed fashion, and each component must fulfill its responsibility to the other components dictated by the architecture.

   Resource allocation decisions also constrain implementation. These decisions may be invisible to implementers working on individual components. The constraints permit a separation of concerns that allows management decisions that make best use of personnel and computational capacity. Component builders must be fluent in the specification of their individual components but not in architectural trade-offs. Conversely, the architects need not be experts in all

© Copyright Virtual University of Pakistan

aspects of algorithm design or the intricacies of the programming language, but they are the ones responsible for architectural trade-offs.

Not only does architecture prescribe the structure of the system being developed, but it also engraves that structure on the structure of the development project. The normal method of dividing up the labor in a large system is to assign different portions of the system to different groups to construct. This is called the work breakdown structure of a system. Because the system architecture includes the highest level decomposition of the system, it is typically used as the basis for the work breakdown structure. Specifically, the module structure is most often the basis for work assignments. The work breakdown structure, in turn, dictates units of planning, scheduling, and budget, as well as inter-team communications channels, configuration control and file system organization, integration and test plans and procedures. Teams communicate with each other in terms of the interface specifications to the major components. The maintenance activity, when launched, will also reflect the software structure, with teams formed to maintain specific structural components.

A side effect of establishing the work breakdown structure is to effectively freeze the software architecture, at least at the level reflected in the work breakdown. A group that is responsible for one of the subsystems will resist having its responsibilities distributed across other groups. If these responsibilities have been formalized in a contractual relationship, changing responsibilities could become expensive. Tracking progress on a collection of tasks that is being distributed would also become much more difficult.

Thus, once the architecture's module structure has been agreed on, it becomes almost impossible for managerial and business reasons to modify it. This is one argument (among many) for carrying out extensive analysis before freezing the software architecture for a large system.

Is it possible to tell that the appropriate architectural decisions have been made (i.e., if the system will exhibit its required quality attributes) without waiting until the system is developed and deployed? If the answer were "no," choosing an architecture would be a hopeless task: random architecture selection would perform as well as any other method. Fortunately, it is possible to make quality predictions about a system based solely on an evaluation of its architecture.

3. **Reusable abstraction of a system.** Software architecture embodies a relatively small, intellectually graspable model for how the system is structured and how its components work together; this model is transferable across systems; in particular, it can be applied to other systems exhibiting similar requirements, and can promote large scale reuse.

   In an architecture-based development of a product line, the architecture is in fact the sum of the early design decisions. System architects choose an architecture

© Copyright Virtual University of Pakistan

(or a family of closely related architectures) that will serve all envisioned members of the product line by making design decisions that apply across the family early and by making other decisions that apply only to individual members late. The architecture defines what is fixed for all members of the family and what is variable.

A family-wide design solution may not be optimal for all systems derived from it, but the quality gained and labor savings realized through architectural-level reuse may compensate for the loss of optimality in particular areas. On the other hand, reusing a family-wide design reduces the risk that a derived system might have an inappropriate architecture. Using a standard design reduces both risk and development costs, at the risk of non-optimality.

8.4 Architectural Attributes

Software architecture must address the non-functional as well as the functional requirements of the software system. This includes performance, security, safety, availability, and maintainability. Following are some of the architectural design guidelines that can help in addressing these challenges.

- Performance
  - Performance can be enhanced by localising operations to minimise sub-system communication. That is, try to have self-contained modules as much as possible so that inter-module communication is minimized.
- Security
  - Security can be improved by using a layered architecture with critical assets put in inner layers.
- Safety
  - Safety-critical components should be isolated
- Availability
  - Availability can be ensured by building redundancy in the system and having redundant components in the architecture.
- Maintainability
  - Maintainability is directly related with simplicity. Therefore, maintainability can be increased by using fine-grain, self-contained components.

8.5 Architectural design process

Just like any other design activity, design of software architecture is a creative and iterative process. This involves performing a number of activities, not necessarily in any particular order or sequence. These include system structuring, control modeling, and modular decomposition. These are elaborated in the following paragraphs.

- System structuring

© Copyright Virtual University of Pakistan

System structuring is concerned with decomposing the system into interacting sub-systems. The system is decomposed into several principal sub-systems and communications between these sub-systems are identified. The architectural design is normally expressed as a block diagram presenting an overview of the system structure. More specific models showing how sub-systems share data, are distributed and interface with each other may also be developed. A sub-system is a system in its own right whose operation is independent of the services provided by other sub-systems. A module is a system component that provides services to other components but would not normally be considered as a separate system.

- Control modelling

  Control modelling establishes a model of the control relationships between the different parts of the system.

- Modular decomposition

  During this activity, the identified sub-systems are decomposed into modules.

This design process is further elaborated in the following section where architectural views are discussed.

© Copyright Virtual University of Pakistan

Lecture No. 23

8.6 Architectural Views

Software architecture defines the high level structure of the software by putting together a number of architectural elements in an organized fashion. These elements are chosen to satisfy the functional as well as non-functional requirements of the system. Perry and Wolfe proposed the following formula for software architecture:

Software architecture = {Elements, Forms, Rationale}

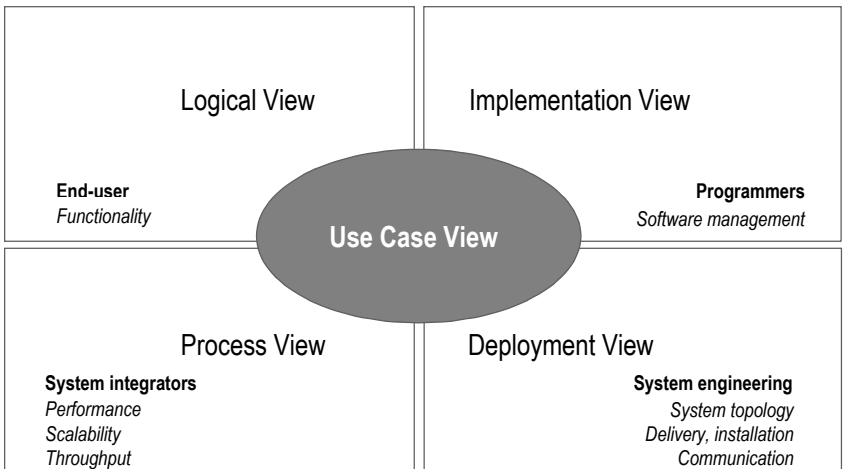
That is, a software architecture is a set of elements which have a certain form. These elements are further divided into three categories. These are: data elements, processing elements, and connecting elements. The data elements contain the information that is used and transformed in the system; the processing elements process the data elements and specify the transformation functions, and connecting elements connect different pieces of architecture together. These can themselves be data or processing elements.

This formula was modified by Boehm as follows:

Software architecture = {Elements, Forms, Rationale/Constraints}

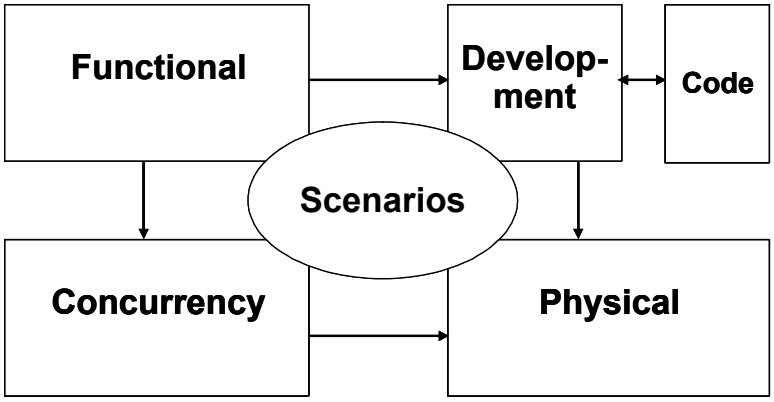
Krutchen proposed that the software architecture model should be composed of multiple views as a software architecture deals with abstraction, with decomposition and composition, with style and esthetics as well as requirements from different stake holders. His architectural model is known as Krutchen's 4+1 architectural view model. As evident, this model proposes the development of 5 main views namely the logical view, the process view, the physical view, the development view, and the use case view. The logical view is the object model of the design, the *process* view captures the concurrency and synchronization aspects of the design, the *physical* view documents the mapping(s) of the software onto the hardware and reflects its distributed aspect, the *development* view describes the static organization of the software in its development environment, and the use case view uses selected use cases or scenarios to validate that the architecture supports the required functionality.

This model is shown in the following diagram.



© Copyright Virtual University of Pakistan

This model has been slightly modified by Clements et. al. and is shown in the following diagram.



Clement's modified version of Krutchen's 4+1 Architectural View Model

In this model, the architecture is again prepared and analyzed from 5 different perspectives. The 4 main views are Functional View, the Concurrency View, the Physical View, and the Development View. Code view is not present in the original Krutchen model and is basically an extension of the development view.

The Functional View comprises of various different functions provided by the system, key system abstraction, and the domain elements. It connects different dependencies and data flows into a single view. This view can be used by the domain engineers, product-line engineers, as well as the end users of the system. It can be used for understanding the functionality provided by the system, modifiability of the system, reusability, tool support, and allocation of work.

The Development View documents the different files and directories in the system and users of this view include the development and the configuration management staff as well as the project managers. The major uses of this view include maintenance, testing, configuration management, and version control.

The components of the Code View include classes, objects, procedures, functions, subsystems, layers, and modules. It documents the calling as well as the containing hierarchy of different components of the system. Its primary users are the programmers and designers of the system. The primary intent of developing this view is to use it for maintenance and portability.

© Copyright Virtual University of Pakistan

Concurrency View intends to document different parallel processes and threads in the system. Its main focus is on event synchronization and parallel data flows. It is used primarily by integrators, performance engineers, and testers. The main purpose of building this view is to identify ways and means to improve performance by highlighting possible opportunities for parallelism.

The Physical View depicts the physical organization of this system and how this system will be physically deployed. This includes different processors, sensors, and storage devices used by the system. This view connects various network elements and communication devices. The primary users of this system include hardware and system engineers. The view is developed with an intention to document and analyze system delivery and installation mechanism. The view is also used in understanding and analyzing issues pertaining to system performance, availability, scalability, and security.

Finally, there are Scenarios. Scenarios are basically use cases which describe the sequences of responsibilities and change cases which are changes to the system. As stated earlier, the first objective of developing a system is to fulfill the functional requirements setout for the system. These functional requirements are written in the form of use cases. Therefore, scenarios are used to understand and validate the system. Their secondary purpose is to communicate the design to the other users of the system. Scenarios tie all the view together into an integrated system. They are also used to understand the dynamic behavior of the system and understand the limits of design.

This is summarized in the following table:

View	Components	Users	Rationale
Functional View	functions, key system abstractions, domain elements	domain engineers, product-line designers, end users	functionality, modifiability, product lines/reusability, tool support, work allocation
Code View	classes, objects, procedures, functions, subsystems, layers, modules	programmers, designers, reusers	modifiability/maintainability, portability, subsetability
Development View	files, directories	managers, programmers, configuration managers	managers, programmers, configuration managers
Physical View	CPUs, sensors, storage	hardware engineers, system engineers	system delivery and installation, performance, availability, scalability, security
Concurrency View	processes, threads	performance engineers, integrators, testers	performance, availability

© Copyright Virtual University of Pakistan

What Are Views Used For?

Views are an engineering tool to help achieve desired system qualities. Each view provides an engineering handle on certain quality attributes. In some systems, distinct views collapse into one (e.g., the concurrency and physical views may be the same for small systems.). Views are also used as documentation vehicle for current development and future development. Users of views include both managers and customers. For these reasons views must be *annotated* to support analysis. This annotation can be achieved with the help of scenarios and design *rationale*.

Structures can also be used to document how the current system was developed and how future development should occur. This information is needed by managers and customers.

Although one often thinks about a system's structure in terms of functionality, there are other system properties in addition to functionality (such as physical distribution, process communication, and synchronization) that must be reasoned about at an architectural level. Each structure provides a method for reasoning about some of the relevant quality attributes. The uses structure, for instance, must be engineered (not merely recorded) to build a system that can easily be extended or contracted. The calls structure is engineered to reduce bottlenecks. The module structure is engineered to produce modifiable systems, and so on. Each structure provides a different view into the system and a different leverage point for design to achieve desired qualities in the system.

Hierarchical Views

Every view is potentially hierarchical, e.g. *functional view could contain* sub-functions. Similarly *development view* contains directories which contain files. C*ode view* would have modules and systems that contain sub-modules and sub-systems respectively. C*oncurrency view* contains processes that are further subdivided into threads. Finally, *physical view* clusters contain computers which contain processors.

Architectural views are related to each other in complicated ways. One has to choose the views that are useful to the system being built and to the achievement of qualities that are important for that particular application. The architectural views should be hierarchical (where needed) and should contain enough annotated information to support desired analyses.

Architectural styles

The architectural model of a system may conform to a generic architectural model or style. An awareness of these styles can simplify the problem of defining system architectures. However, most large systems are heterogeneous and do not follow a single architectural style.

© Copyright Virtual University of Pakistan

Each style describes a system category that encompasses:

(1) a set of components (e.g., a database, computational modules) that perform a function required by a system,
(2) a set of connectors that enable "communication, coordination and cooperation" among components,
(3) constraints that define how components can be integrated to form the system, and

semantic models that enable a designer to understand the overall properties of a system by analyzing the known properties of its constituent parts.

Lecture No. 24

8.7 Architectural models

Like analysis models, many different kinds of architectural models are developed during the architectural design process. Static structural model shows the major system components while a dynamic process model shows the process structure of the system. Interface models are developed to define sub-system interfaces.

8.8 Architectural Styles

Architectural design may be based upon a certain pattern of model. These different patterns are also called architectural styles. These styles have different characteristics and attributes and can be useful to solve problems related to a particular situation of requirement. Among the many styles, the most commonly practiced are the following:

- Data-centered architectures
- Client Server Architecture and its variations
- Layered architectures
- Reference Architecture

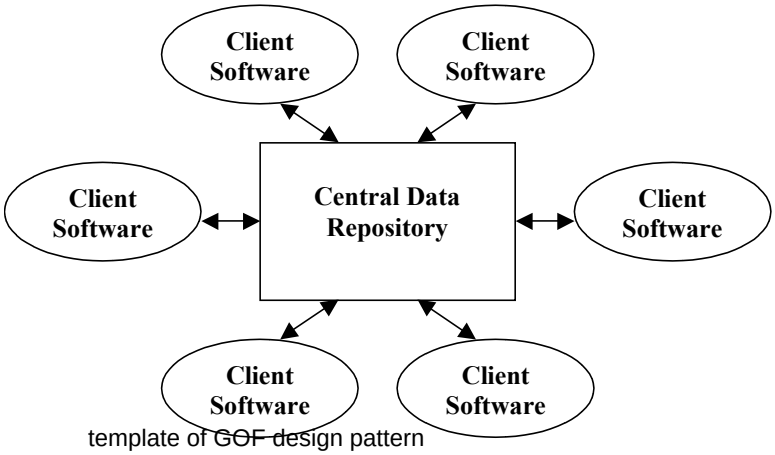
Data-Centered or the repository model

In any system, sub-systems need to exchange information and data. This may be done in two ways:

1. Shared data is held in a central database or repository and may be accessed by all sub-systems
2. Each sub-system maintains its own database and passes data explicitly to other sub-systems

© Copyright Virtual University of Pakistan

When large amounts of data are to be shared, the repository model of sharing is most commonly used. This model has been extensively used in the main-frame based



template of GOF design pattern includes

application. The model is depicted in the following diagram:

Repository model characteristics

- Advantages

  Repository model is an efficient way to share large amounts of data. In this case sub-systems need not be concerned with how data is produced Centralised management e.g. backup, security, etc. This model also provides a global view of the system and the sharing model is published as the repository schema.

- Disadvantages

  Repository model suffers from a number of disadvantages. First of all, sub-systems must agree on a repository data model. This inevitably leads to a compromise. The major problem however is that data evolution is difficult and expensive. There is also little scope for implementing specific management policies. It is also difficult to distribute efficiently

8.9 Client-server model

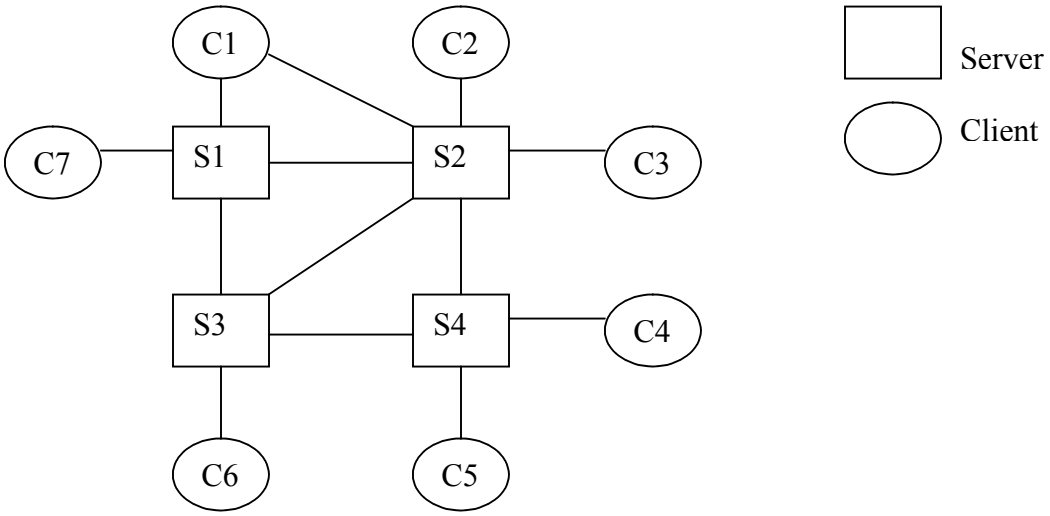
Client server model tries to distribute data and processing. This is a shift from main-frame based applications where both the data management and the processing of data used to be typically carried out by the same main-frame computer. In those applications, the user interface used to be a provided through a "dumb" terminal which did not have much processing power. With the availability of the cheaper but power machines, it was possible to shift some load from the back-end computer to other smaller machines.

The client-server model is a distributed system model which shows how data and processing is distributed across a range of components. In this model, the application is modeled as a set of services that are provided by servers and a set of clients that use these services. The system is organized as a set of stand-alone servers which provide specific services such as printing, data management, etc. and a set of clients which call on these

© Copyright Virtual University of Pakistan

services. These clients and servers are connected through a network which allows clients to access servers. Clients and servers are logical processes (not always physical machines). Clients know the servers but the servers do not need to know all the clients and the mapping of processes to processors is not always 1:1.

The following diagram depicts a general client-server organization.



© Copyright Virtual University of Pakistan

8.10Client/Server Software Components

A typical client-server architecture based system is composed of a number of different components. These include user interaction/presentation subsystem, application subsystem, database management subsystem, and middleware. The application subsystem implements requirements defined by the application within the context of the operating environment. In this case the application components may reside on either client or server side. Middleware provides the mechanism and protocols to connect clients with the servers.

Representative Client/Server Systems

Following are some of the representative server types in a client-server systems.

- File servers
  In this case, client requests selected records from a file and the server transmits records to client over the network.
- Database servers
  In this case, client sends requests, such as SQL queries, to the database server, the server processes the request and returns the results to the client over the network.
- Transaction servers
  In this configuration, client sends requests that invokes remote procedures on the server side, server executes procedures invoked and returns the results to the client.
- Groupware servers
  Groupware servers provide set of applications that enable communication among clients using text, images, bulletin boards, video, etc.

Client-server characteristics

- Advantages
  The main advantage of the client-server architecture is that it makes effective use of networked systems. Instead of building the system with very expensive large computers and hardware devices, cheaper hardware may be used to gain performance and scalability. In addition, adding new servers or upgrading existing servers becomes easier. Finally, distribution of data is straightforward in this case.

- Disadvantages
  The main disadvantage of this model is that there is no standard way of sharing data, so sub-systems may use different data organisation. Hence data interchange may be inefficient. From a management point of view, each server needs attention and hence there is redundant management in each server. Finally there is no central register of names and services - it may be hard to find out what servers and services are available.

© Copyright Virtual University of Pakistan

8.11 Representative Client/Server Configurations

The client-server model can have many different configurations. In the following sections, we look at some of these configurations.

Thin Client Model

This model was initially used to migrate legacy systems to client server architectures. In this case the legacy system may act as a server in its own right and the GUI may be implemented on a client. It chief disadvantage is that it places a heavy processing load on both the server and the network.

Fat Client Model

With advent of cheaper and more powerful hardware, people thought of using the processing power of client side machines. So the fat client model came into being. In this model, more processing is delegated to the client as the application processing is locally extended. It is suitable for new client/server systems when the client system capabilities are known in advance. It however is more complex than thin client model with respect to management issues. Since the client machine now also has a significant part of the application resident on it, new versions of each application need to be installed on every client.

Lecture No. 25

Zero Install

As discussed earlier, fat-client architecture posed major challenges in terms of installation and maintenance of the client side of the application, especially when there are large number of client machines. So the idea behind zero install architecture is to develop a system where no installation on the client side is needed. This can only be done when there is no or little processing done at the client side. This is basically a trade-off between using the computing power available at the client machine versus maintenance overhead. This in essence takes us back to an architecture which is similar to thin-client architecture. There is little difference though. In the classical thin-client architecture, the entire processing is carried-out by a single server, in the case of zero-install, the network environment is used to distribute server side processing by adding a number of servers which share processing load. Web-based application where the web-pages are put on a web-server is an example of this type of architecture. In this case, whenever there is a change, the web page is updated accordingly. Now when the user logs in, he gets to use the modified version of the application without any changes or updates at the client side.

© Copyright Virtual University of Pakistan

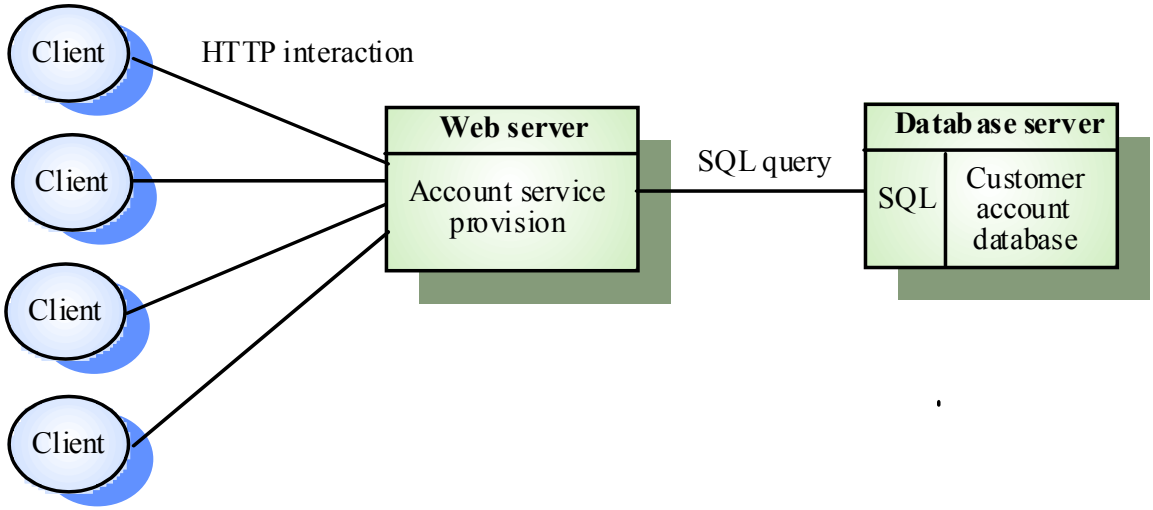
N-Tier architecture

N-tier architecture stems from the struggle to find a middle ground between the fat-client architecture and the thin-client architecture. In this case the idea is to enhance scalability and performance by distributing both the data and the application using multiple server machines. This could involve different types of servers such as application server, web server, and DB server. Three-tier architecture which is explained below is a specialized form of this architecture.

Three-tier Architecture

In this architecture, each application architecture layers (presentation, application, database) may run on separate processors. It therefore allows for better performance than a thin-client approach. It is simpler to manage than fat client approach and highly scalable (as demands increase add more servers).

A typical 3-tier architecture is depicted in the following diagram.



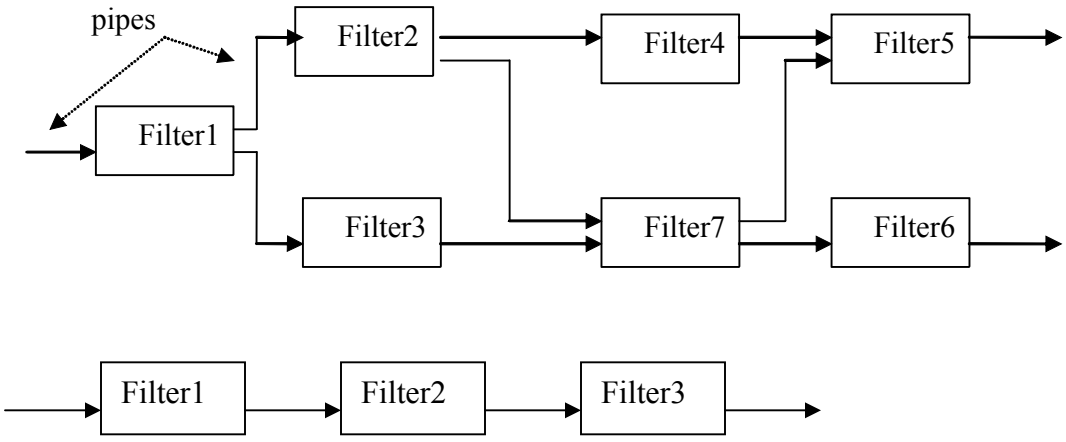
N-tier architecture generalizes the concepts of 3-tier architecture. In this case the system architecture may have more than 3 layers. That is, in n-tier architecture, in order to increase performance, we may distribute the application over different servers by putting different subsystems on different servers.

© Copyright Virtual University of Pakistan

8.12 Data Flow or Pipes and Filters Architecture

This architecture is very similar to data flow diagrams. This is used when the input data is processed through a series of transformations to yield the desired output. It is also known as pipes and filters architecture where each processing step is called a filter and the connecting link from one process to the other is called the pipe through which the information flows from one process to the other. An important aspect of this model is that each filter works independently of others and does not require knowledge of the working of any of the other filters including its neighbours.

If the dataflow has only a single sequence of processes with no alternative or parallel paths, then it is called batch sequential. These models are depicted in the following diagrams.

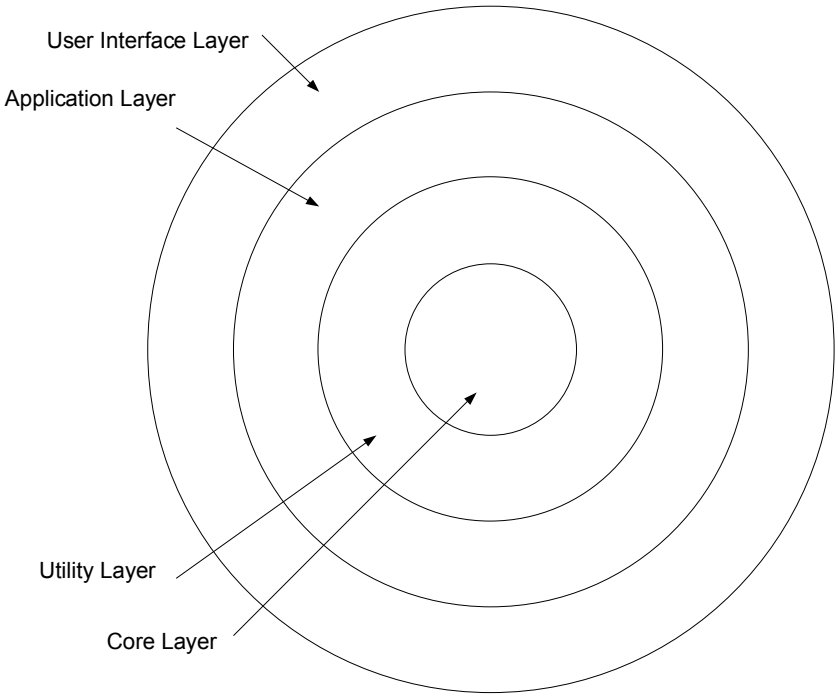


Batch Sequential

© Copyright Virtual University of Pakistan

Layered Architecture

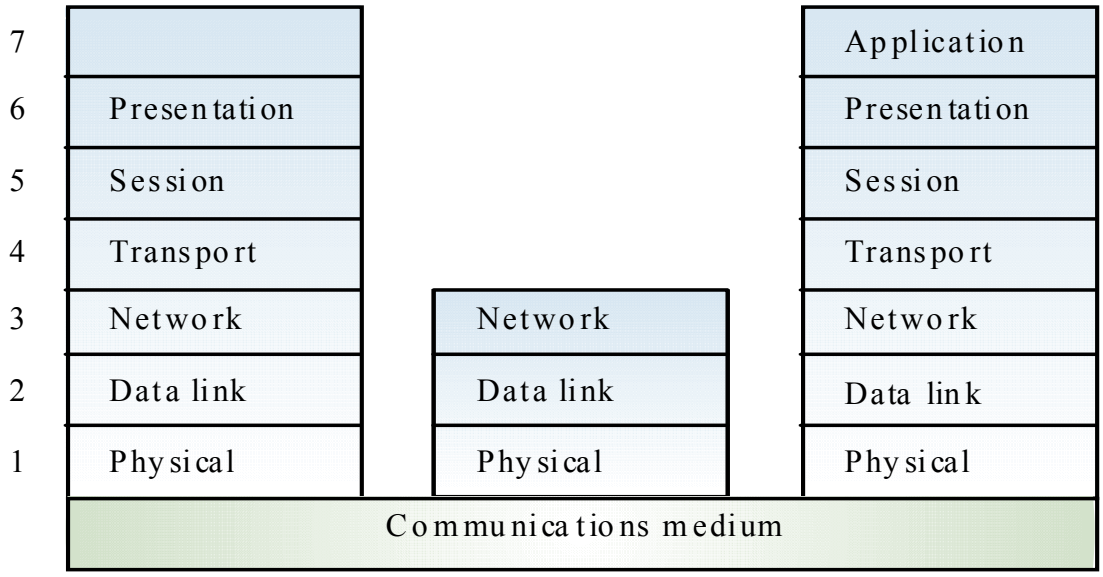
As the name suggests, a layered architecture has many different layers. One typical example of a layered architecture is an operating system where different layers are used to provide services and functionality and the inner layers are closer to the machine hardware than the outer layers. In this way, each layer isolates the outer layer from inner complexities. In order to work properly, the outer layer only needs to know the interface provided by the inner layer. If there are any changes in the inner layer, as long as the interface does not change, the outer layer is not affected. This scheme tremendously portability of the system. The basic layered architecture is depicted in the following diagram.



© Copyright Virtual University of Pakistan

8.13 Reference architectures

Reference architecture is not a physical architecture. It is only a reference for defining protocols and designing and implementing systems developed by different parties. Reference models are derived from a study of the application domain rather than from existing systems. It may be used as a basis for system implementation or to compare different systems. It acts as a standard against which systems can be evaluated. One very common example of such a reference model is the OSI model which is a layered model for communication systems. The success of this kind model is evident from the success of the Internet where all kinds of heterogeneous systems can talk to each other only because all of them use the same reference architecture.



OSI reference model

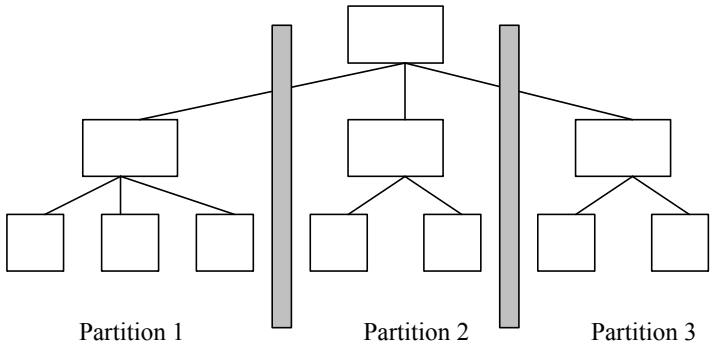
© Copyright Virtual University of Pakistan

8.14 Partitioning the Architecture

Partitioning of architecture is an important concept. What we basically want to do is distribute the responsibilities to different subsystems so that we get a software system which is easy to maintain. Partitioning results in a system that suffers from fewer side effects. This ultimately means that we get a system that is easier to test and extend and hence is easier to maintain.

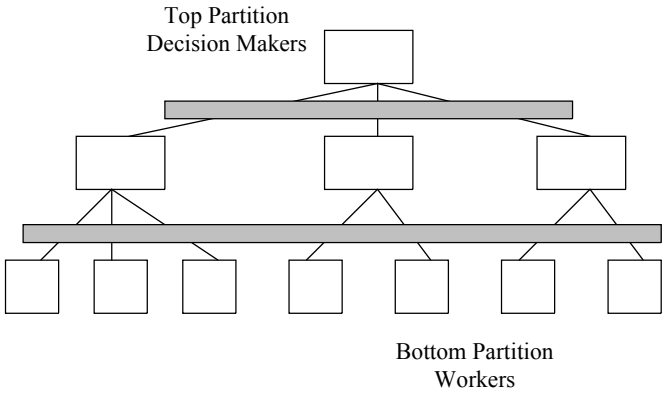
Partitioning of an architecture may be "horizontal" and/or "vertical".

In the horizontal partitioning we define separate branches of the module hierarchy for each major function and control modules are used to coordinate communication between functions. This concept is depicted in the following diagram.



Vertical partitioning divides the application from a decision making perspective. The architecture is partitioned in horizontal layers so that decision making and work are stratified with the decision making modules residing at the top of the hierarchy and worker coming at the bottom. This partitioning is also known as factoring and the general model is depicted in the following diagram.

© Copyright Virtual University of Pakistan



8.15 Analyzing Architecture design

In a given system, the required characteristics may conflict. Trade-offs seek optimal combinations of properties based on cost/benefit analysis. So the analysis requires an understanding of what is required and the relative priority of the attributes has to be established. The following sequence of steps provides a guideline for performing architectural analysis.

1. Collect scenarios.
2. Elicit requirements, constraints, and environment description.
3. Describe the architectural styles/patterns that have been chosen to address the scenarios and requirements. These views include module view, process view, and data flow view.
4. Evaluate quality attributes by considered each attribute in isolation.
5. Identify the sensitivity of quality attributes to various architectural attributes for a specific architectural style.

Critique candidate architectures (developed in step 3) using the sensitivity analysis conducted in step 5.

© Copyright Virtual University of Pakistan

Lecture No. 26

Introduction to Design Patterns

Design Patterns

Christopher Alexander says, "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice." A Pattern Language: Towns/Buildings/Construction, 1977

Even though Alexander was talking about patterns in buildings and towns, what he says is true about object-oriented design patterns. Our solutions are expressed in terms of objects and interfaces instead of walls and doors, but the core of both kinds of patterns is a solution to a problem in a context.

Design Patterns defined

"Description of communicating objects and classes that are customized to solve a general design in a particular context."

Patterns are devices that allow programs to share knowledge about their design. In our daily programming, we encounter many problems that have occurred, and will occur again. The question we must ask our self is how we are going to solve it *this* time. Documenting patterns is one way that you can reuse and possibly share the information that you have learned about how it is best to solve a specific program design problem.

Essay writing is usually done in a fairly well defined form, and so is documenting design patterns. The general form for documenting patterns is to define items such as:

- The motivation or context that this pattern applies to.
- Prerequisites that should be satisfied before deciding to use a pattern.
- A description of the program structure that the pattern will define.
- A list of the participants needed to complete a pattern.
- Consequences of using the pattern...both positive and negative.
- Examples!

Historical perspective of design patterns

The origin of design patterns lies in work done by an architect named *Christopher Alexander* during the late 1970s. He began by writing two books, *A Pattern*

© Copyright Virtual University of Pakistan

Language[Alex77] and *A Timeless Way of Building* [Alex79] which, in addition to giving examples, described his rationalle for documenting patterns.

The pattern movement became very quiet until 1987 when patterns appeared again at an OOPSLA conference. Since then, many papers and presentations have appeared, authored by people such as Grady Booch, Richard Helm, and Erich Gamma, and Kent Beck. From then until 1995, many periodicals, featured articles directly or indirectly relating to patterns. In 1995, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides published Design Patterns: Elements of Reusable Object-Oriented Software [Gamma95], which has been followed by more articles in trade journals.

The concept of design patterns is not new as we can find a number of similar pursuits in the history of program designing and writing. For instance, Standard Template Library (STL) is a library of reusable components provided by C++ compilers. Likewise, we use algorithms in data structures that implement typical operations of manipulating data in data structures. Another, similar effort was from Peter Coad whose patterns are known for object-oriented analysis and design.
Anti-patterns is another concept that corresponds to common mistakes in analysis and design. These are identified in order to prevent potential design and analysis defects from entering into the design. Another, similar concept is object-oriented framework that is a set of cooperative classes that make up reusable design of a system. Framework dictates the architecture of the software and describes the limitations and boundaries of architecture.
With this introduction, we now describe the format that has been adopted in GoF book for describing various design patterns.

GOF Design Pattern Format

The basic template includes ten things as described below

Name

- Works as idiom
- Name has to be meaningful

Problem

- A statement of the problem which describes its intent
- The goals and objectives it wants to reach within the given context

Context

- Preconditions under which the problem and its solutions seem to occur
- Result or consequence
- State or configuration after the pattern has been applied

Forces

- Relevant forces and constraints and their interactions and conflicts.
- motivational scenario for the pattern.

Solution

- Static and dynamic relationships describing how to realize the pattern.
- Instructions on how to construct the work products.

© Copyright Virtual University of Pakistan

- Pictures, diagrams, prose which highlight the pattern's structure, participants, and collaborations.

Examples

- One or more sample applications to illustrate
  - a specific context
  - how the pattern is applied
-

Resulting context

- the state or configuration after the pattern has been applied
- consequences (good and bad) of applying the pattern

Rationale

- justification of the steps or rules in the pattern
- how and why it resolves the forces to achieve the desired goals, principles, and philosophies
- how are the forces orchestrated to achieve harmony
- how does the pattern actually work

Related patterns

- the static and dynamic relationships between this pattern and other patterns

Known uses

- to demonstrate that this is a proven solution to a recurring problem

Classifications of deerns

Creational patterns

- How to create and instantiate
- Abstract the instantiation process and make the system independent of its creational process.
- Class creational rules
- Object creational rules
- Abstract factory and factory method
- Abstract the instantiation process
- Make a system independent to its realization
- Class Creational use inheritance to vary the instantiated classes
- Object Creational delegate instantiation to an another object

Structural patterns

- Deals with object's structure
- Class structural patterns concern the aggregation of classes to form the largest classes.
- Object structural patterns concerns the aggregation of objects to form the largest classes
- Class Structural patterns concern the aggregation of classes to form largest structures
- Object Structural pattern concern the aggregation of objects to form largest structures

Behavioral patterns

© Copyright Virtual University of Pakistan

- Describe the patterns of communication between classes and objects
- How objects are communicating with each other
- Behavioral class patterns
- Behavioral object patterns
- Use object composition to distribute behavior between classes
- Help in distributing object's intelligence
- Concern with algorithms and assignment of responsibilities between objects
- Describe the patterns of communication between classes or objects
- Behavioral class pattern use inheritance to distribute behavior between classes
- Behavioral object pattern use object composition to distribute behavior between classes

In the following, one pattern from each of the above mentioned categories of design patterns is explained on GoF format.

Lecture No. 27

Observer Pattern

Name

- Observer

Basic intent

- It is intended to define a many to many relationship between objects so that when one object changes state all its dependants are notified and updated automatically.
- Dependence/publish-subscribe mechanism in programming language
    - Smalltalk being the first pure Object Oriented language in which observer pattern was used in implementing its Model View Controller (MVC) pattern. It was a publish-subscribe mechanism in which views (GUIs) were linked with their models (containers of information) through controller objects. Therefore, whenever underlying data changes in the model objects, the controller would notify the view objects to refresh themselves and vice versa.
    - MVC pattern was based on the observer pattern.

Motivation

- It provides a common side effect of partitioning a system into a collection of cooperating classes that are in the need to maintain consistency between related objects.

Description

- This can be used when multiple displays of state are needed.

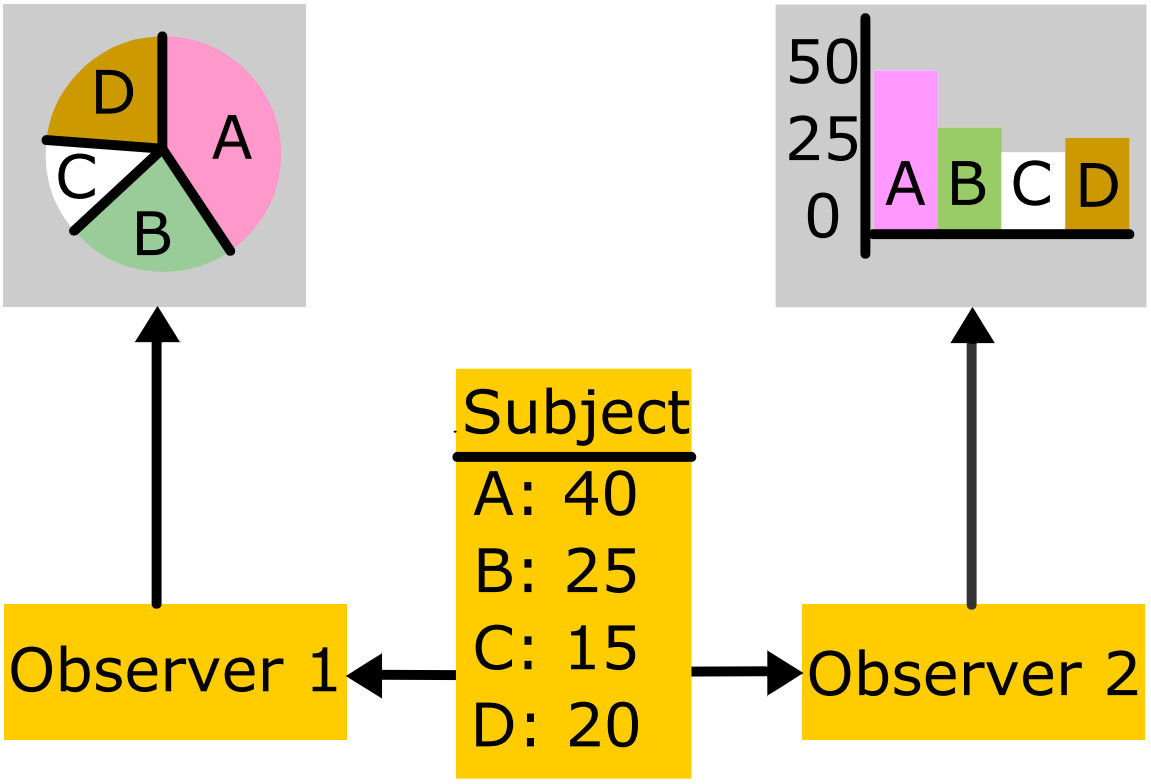
Consequences

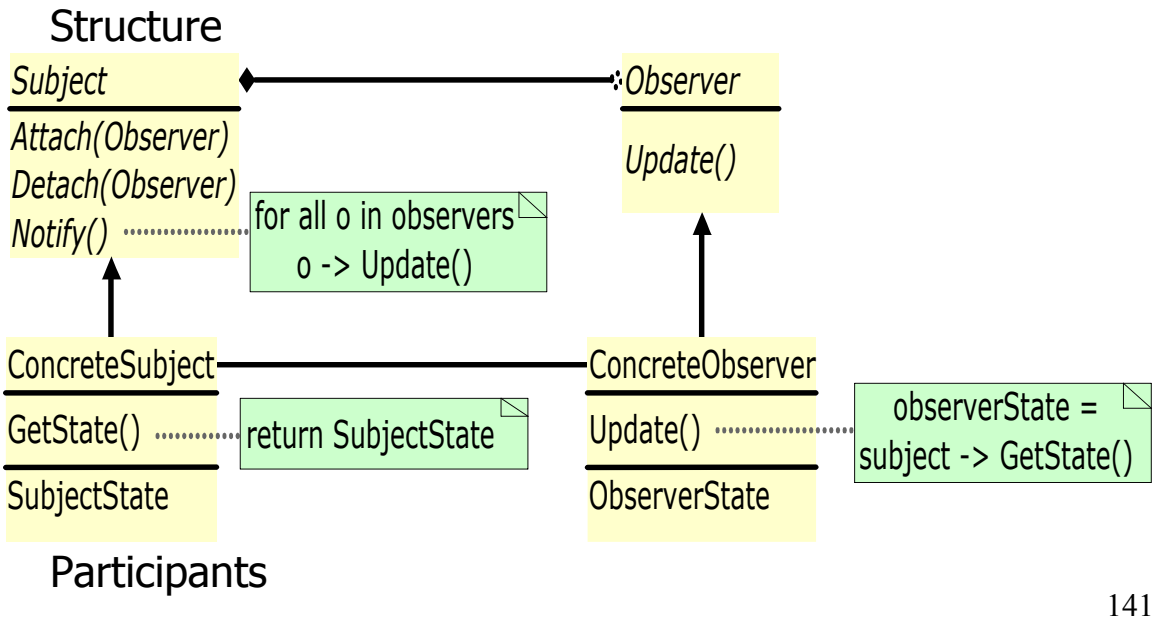
- **Optimizations to enhance display performance are impractical.**

© Copyright Virtual University of Pakistan

Example implementation of Observer PatternObject Model

Many graphical user interface toolkits separate the presentational aspects of the user interface from the underlying application data. Classes defining application data and presentations can be reused independently. They can work together, too. Both a spreadsheet object and bar chart object can depict information in the same application data object using different presentations. The spreadsheet and the bar chart don't know about each other, thereby letting you reuse only the one you need. But they behave as though they do. When the user changes the information in the spreadsheet, the bar chart reflects the changes immediately, and vice versa.





© Copyright Virtual University of Pakistan

Subject

- Knows its observers. Any number of Observer objects may observe a subject.
- Provides an interface for attaching and detaching Observer objects.

Observer

- Defines an updating interface for objects that should be notified of changes in a subject.

ConcreteSubject

- Stores state of interest to concreteObserver objects.
- Sends a notification to its observers when its state changes.

ConcreteObserver

- Maintains a reference to a ConcreteSubject object.
- Stores state state that should stay consistent with the subject's.
- Implements the Observer updating interface to keep its state consistent with the subject's.

Singleton Pattern

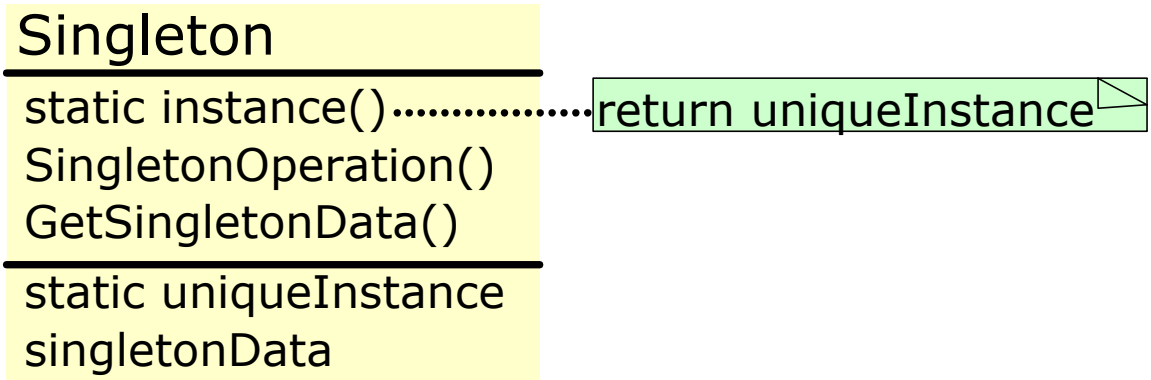
Intent

- It ensures that a class only has one instance and provides a global point of access to it.

Applicability

- Singleton pattern should be used when there must be exactly one instance of a class and it must be accessible to clients from a well-known access point.
- Singleton pattern should be used when controlling the total number of instances that would be created for a particular class.
- Singleton pattern should be used when the sole instance should be extensible by sub classing and clients should be able to use an extended instance without modifying their code.

Structure



Participants

Singleton

© Copyright Virtual University of Pakistan

- Defines an instance operation that lets clients access its unique instance. Instance is a class operation (that is, a class method in Smalltalk and a static member function in C++).
- May be responsible for creating its own unique instance.

Singleton Pattern Example

The Singleton class is declared as

```
class Singleton {
public:
    static Singleton* Instance();
protected:
    Singleton();
private:
    static Singleton* _instance;
};
```

The corresponding implementation is

```
Singleton* Singleton::_instance = 0;

Singleton* Singleton::Instance(){
     if (_instance == 0) {
        _instance = new Singleton;
     }
     return _instance;
}
```

Clients access the singleton exclusively through the Instance member function. The variable _instance is initialized to 0, and the static member function Instance returns its value, initializing it with the unique instance if it is 0.

Façade Pattern

Intent

- It provides a unified interface to a set of interfaces in a sub-system.
- Façade defines a higher level interface that makes a subsystem easier to use

Applicability

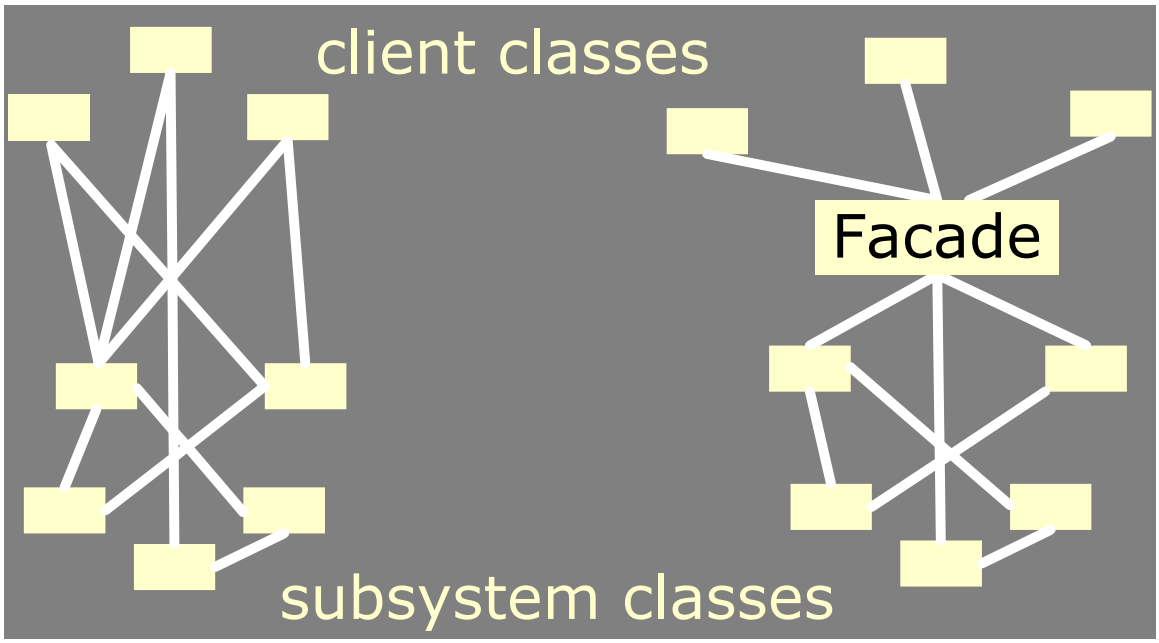
- You would use façade when you want to provide a simple interface to a complex sub-system.
- You would use façade pattern when there are many dependencies between clients and the implementation classes of an abstraction.
- You should introduce a façade to decouple the system from clients and other subsystems.
- You want to layer your subsystem.
- You would use façade when you want to provide a simple interface to a complex sub-system

© Copyright Virtual University of Pakistan

- You would use façade pattern when there are many dependencies between clients and the implementation classes of an abstraction
- You should introduce a façade to decouple the system from clients and other subsystems
- You want to layer the subsystem

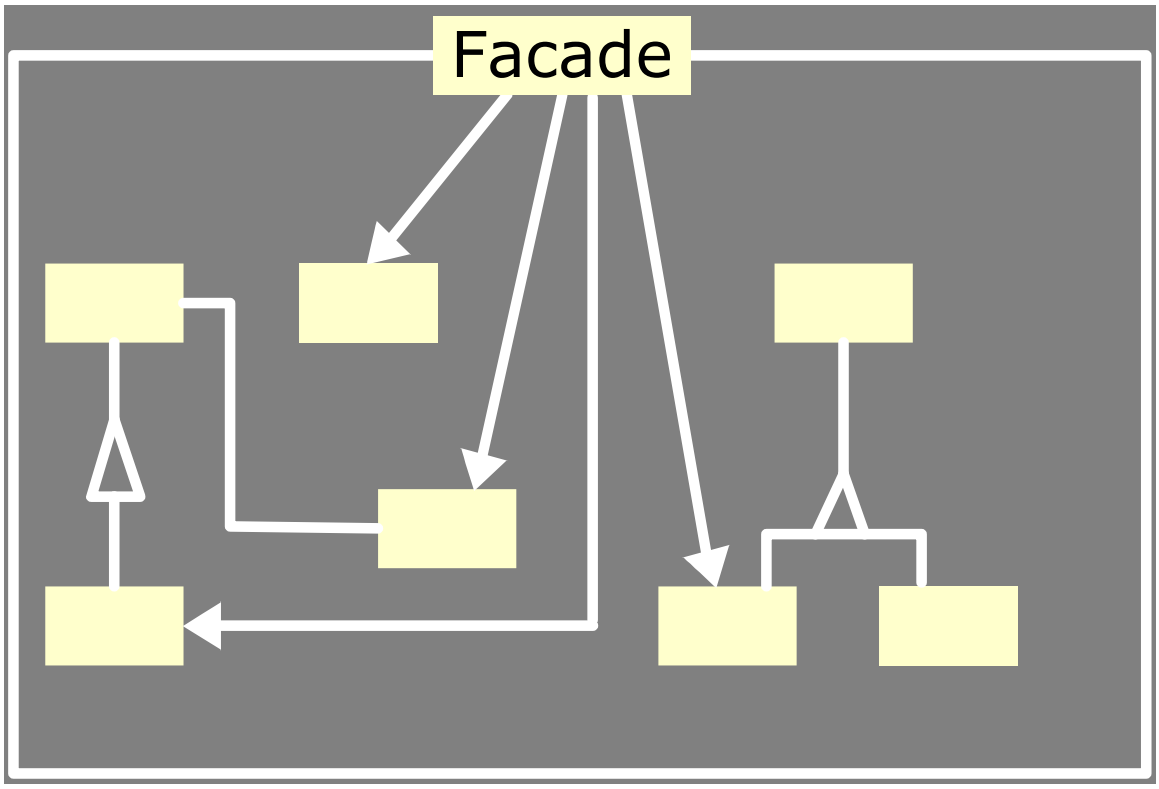
Abstract example of façade

Structuring a system into subsystems helps reduce complexity. A common design goal is to minimize the communication and dependencies between subsystems. One way to achieve this goal is to introduce a façade object that provides a single, simplified interface to the more general facilities of a subsystem.



Structure

© Copyright Virtual University of Pakistan



Participants

Façade

- Knows which subsystem classes are responsible for a request.
- Delegates client requests to appropriate subsystem objects.

Subsystem classes

- Implement subsystem functionality.
- Handle work assigned by the Façade object.
- Have no knowledge of the façade, that is, they keep no references to it.

© Copyright Virtual University of Pakistan

Lecture No. 28

Good Programming Practices and Guidelines

10.1 Maintainable Code

As we have discussed earlier, in most cases, maintainability is the most desirable quality of a software artifact. Code is no exception. Good software ought to have code that is easy to maintain. Fowler says, "*Any fool can write code that computers can understand, good programmers write code that humans can understand."* That is, it is not important to write code that works, it is important to write code that works and is easy to understand so that it can be maintained. The three basic principles that guide maintainability are: simplicity, clarity, and generality or flexibility. The software will be easy to maintain if it is easy to understand and easy to enhance. Simplicity and clarity help in making the code easier to understand while flexibility facilitates easy enhancement to the software.

© Copyright Virtual University of Pakistan

Self Documenting Code

From a maintenance perspective, what we need is what is called ***self documenting code***. A *self documenting code* is a code that explains itself without the need of comments and extraneous documentation, like flowcharts, UML diagrams, process-flow state diagrams, etc. That is, the meaning of the code should be evident just by reading the code without having to refer to information present outside this code.

The question is: how can we write code that is self-documenting?

There are a number of attributes that contributes towards making the program self documented. These include, the size of each function, choice of variable and other identifier names, style of writing expressions, structure of programming statements, comments, modularity, and issues relating to performance and portability.

The following discussion tries to elaborate on these points.

Function Size

The size of individual functions plays a significant role in making the program easy or difficult to understand. In general, as the function becomes longer in size, it becomes more difficult to understand. Ideally speaking, a function should not be larger than 20 lines of code and in any case should not exceed one page in length. From where did I get this number 20 and one page? The number 20 is approximately the lines of code that fit on a computer screen and one page of course refers to one printed page. The idea behind these heuristics is that when one is reading a function, one should not need to go back and forth from one screen to the other or from one page to the other and the entire context should be present on one page or on one screen.

© Copyright Virtual University of Pakistan

Identifier Names

Identifier names also play a significant role in enhancing the readability of a program. The names should be chosen in order to make them meaningful to the reader. In order to understand the concept, let us look at the following statement.

```
if (x==0)   // this is the case when we are allocating a new number
```

In this particular case, the meanings of the condition in the if-statement are not clear and we had to write a comment to explain it. This can be improved if, instead of using x, we use a more meaningful name. Our new code becomes:

```
if (AllocFlag == 0)
```

The situation has improved a little bit but the semantics of the condition are still not very clear as the meaning of 0 is not very clear. Now consider the following statement:

```
If (AllocFlag == NEW_NUMBER)
```

We have improved the quality of the code by replacing the number 0 with a named constant NEW_NUMBER. Now, the semantics are clear and do not need any extra comments, hence this piece of code is self-documenting.

© Copyright Virtual University of Pakistan

10.2 Coding Style Guide

Consistency plays a very important role in making it self-documenting. A consistently written code is easier to understand and follow. A coding style guide is aimed at improving the coding process and to implement the concept of standardized and relatively uniform code throughout the application or project. As a number of programmers participate in developing a large piece of code, it is important that a consistent style is adopted and used by all. Therefore, each organization should develop a style guide to be adopted by its entire team.

This coding style guide emphasizes on C++ and Java but the concepts are applicable to other languages as well.

10.3 Naming Conventions

Hungarian Notation was first discussed by Charles Simonyi of Microsoft. It is a variable naming convention that includes information about the variable in its name (such as data type, whether it is a reference variable or a constant variable, etc). Every company and programmer seems to have their own flavor of Hungarian Notation. The advantage of Hungarian notation is that by just looking at the variable name, one gets all the information needed about that variable.

Bicapitalization or *camel case* (frequently written *CamelCase*) is the practice of writing compound words or phrases where the terms are joined without spaces, and every term is capitalized. The name comes from a supposed resemblance between the bumpy outline of the compound word and the humps of a camel. CamelCase is now the official convention for file names and identifiers in the Java Programming Language.

In our style guide, we will be using a naming convention where Hungarian Notation is mixed with CamelCase.

© Copyright Virtual University of Pakistan

General Naming Conventions

General naming conventions for Java and C++

1. Names representing types must be nouns and written in mixed case starting with upper case.

```
Line, FilePrefix
```

2. Variable names must be in mixed case starting with lower case.

```
line, filePrefix
```

This makes variables easy to distinguish from types, and effectively resolves potential naming collision as in the declaration `Line line;`

3. Names representing constants must be all uppercase using underscore to separate words.

```
MAX_ITERATIONS, COLOR_RED
```

In general, the use of such constants should be minimized. In many cases implementing the value as a method is a better choice. This form is both easier to read, and it ensures a uniform interface towards class values.

```
int    getMaxIterations()//    NOT:    MAX_ITERATIONS    =    25
{
                              return                          25;
}
```

4. Names representing methods and functions should be verbs and written in mixed case starting with lower case.

```
getName(), computeTotalWidth()
```

5. Names representing template types in C++ should be a single uppercase letter.

```
template<class                        T>                      ...
template<class C, class D> ...
```

6. Global variables in C++ should always be referred to by using the :: operator.
::mainWindow.open() , ::applicationContext.getName()

7. Private class variables should have _ suffix.

```
class                                                    SomeClass
{
                private              int                 length_;
                                                              ...
}
```

Apart from its name and its type, the *scope* of a variable is its most important feature. Indicating class scope by using _ makes it easy to distinguish class variables from local scratch variables.

© Copyright Virtual University of Pakistan

8. Abbreviations and acronyms should not be uppercase when used as name.

```
exportHtmlSource();          //    NOT:    xportHTMLSource();
openDvdPlayer();       // NOT: openDVDPlayer();
```

Using all uppercase for the base name will give conflicts with the naming conventions given above. A variable of this type would have to be named dVD, hTML etc. which obviously is not very readable.

9. Generic variables should have the same name as their type.

```
void setTopic (Topic topic)        // NOT: void setTopic (Topic value)
                                   // NOT: void setTopic (Topic aTopic)
                                       // NOT: void setTopic (Topic x)

void connect (Database database)  // NOT: void connect (Database db)
                                 // NOT: void connect (Database oracleDB)
```

Non-generic variables have a *role*. These variables can often be named by combining role and type:

```
Point              startingPoint,              centerPoint;
Name  loginName;
```

10. All names should be written in English.

```
fileName;    // NOT:   filNavn
```

11. Variables with a large scope should have long names, variables with a small scope can have short names. Scratch variables used for temporary storage or indices are best kept short. A programmer reading such variables should be able to assume that its value is not used outside a few lines of code. Common scratch variables for integers are *i*, *j*, *k*, *m*, *n* and for characters *c* and *d*.

12. The name of the object is implicit, and should be avoided in a method name.

```
line.getLength();    // NOT:  line.getLineLength();
```

The latter seems natural in the class declaration, but proves superfluous in use.

© Copyright Virtual University of Pakistan

Specific Naming Conventions for Java and C++

1. The terms *get/set* must be used where an attribute is accessed directly.

```
employee.getName();
matrix.getElement (2, 4);
employee.setName (name);
matrix.setElement (2, 4, value);
```

2. is prefix should be used for boolean variables and methods.

```
isSet, isVisible, isFinished, isFound, isOpen
```

Using the *is* prefix solves a common problem of choosing bad boolean names like *status* or *flag*. *isStatus* or *isFlag* simply doesn't fit, and the programmer is forced to chose more meaningful names.

There are a few alternatives to the *is* prefix that fits better in some situations. These are *has*, *can* and *should* prefixes:

```
boolean hasLicense();
boolean canEvaluate();
boolean shouldAbort = false;
```

3. The term *compute* can be used in methods where something is computed.

```
valueSet.computeAverage();  matrix.computeInverse()
```

Using this term will give the reader the immediate clue that this is a potential time consuming operation, and if used repeatedly, he might consider caching the result.

4. The term *find* can be used in methods where something is looked up.

```
vertex.findNearestVertex();   matrix.findMinElement();
```

This gives the reader the immediate clue that this is a simple look up method with a minimum of computations involved.

5. The term *initialize* can be used where an object or a concept is established.

```
printer.initializeFontSet();
```

6. List suffix can be used on names representing a list of objects.

`vertex` (one vertex), `vertexList` (a list of vertices)

Simply using the plural form of the base class name for a list (`matrixElement` (one matrix element), `matrixElements` (list of matrix elements)) should be avoided since the two only differ in a single character and are thereby difficult to distinguish.
A *list* in this context is the compound data type that can be traversed backwards, forwards, etc. (typically a `Vector`). A plain array is simpler. The suffix *Array* can be used to denote an array of objects.

7. n prefix should be used for variables representing a number of objects.

```
nPoints, nLines
```


© Copyright Virtual University of Pakistan

The notation is taken from mathematics where it is an established convention for indicating a number of objects.

8. No suffix should be used for variables representing an entity number.

```
tableNo, employeeNo
```

The notation is taken from mathematics where it is an established convention for indicating an entity number. An elegant alternative is to prefix such variables with an *i*: `iTable, iEmployee`. This effectively makes them *named* iterators.

9. Iterator variables should be called ***i***, ***j***, ***k*** etc.

```
while (Iterator i = pointList.iterator(); i.hasNext(); ) {
  :
}

for (int i = 0; i < nTables; i++) {
  :
}
```

The notation is taken from mathematics where it is an established convention for indicating iterators. Variables named *j*, *k* etc. should be used for nested loops only.

10. Complement names must be used for complement entities.

get/set, add/remove, create/destroy, start/stop, insert/delete, increment/decrement, old/new, begin/end, first/last, up/down, min/max, next/previous, old/new, open/close, show/hide
Reduce complexity by symmetry.

11. Abbreviations in names should be avoided.

```
computeAverage();      // NOT:  compAvg();
```

There are two types of words to consider. First are the common words listed in a language dictionary. These must never be abbreviated. Never write:

`cmd`	instead of	`command`
`cp`	instead of	`copy`
`pt`	instead of	`point`
`comp`	instead of	`compute`
`init`	instead of	`initialize`

etc.

Then there are domain specific phrases that are more naturally known through their acronym or abbreviations. These phrases should be kept abbreviated. Never write:

`HypertextMarkupLanguage`	instead of	`html`
`CentralProcessingUnit`	instead of	`cpu`
`PriceEarningRatio`	instead of	`pe`

etc.

12. Negated boolean variable names must be avoided.

© Copyright Virtual University of Pakistan

```
boolean isError;     // NOT:   isNotError
boolean isFound;     // NOT:   isNotFound
```

The problem arise when the logical **NOT** operator is used and double negative arises. It is not immediately apparent what `!isNotError` means.

13. Functions (methods returning an object) should be named after what they return and procedures (*void* methods) after what they do. This increases readability. Makes it clear what the unit should do and especially all the things it is *not* supposed to do. This again makes it easier to keep the code clean of side effects. Naming pointers in C++ specifically should be clear and should represent the pointer type distinctly.

```
Line *line          //NOT    Line *pLine; or Line *lineptr; etc
```


© Copyright Virtual University of Pakistan

Lecture No. 29

10.4 File handling tips for Java and C++

1. C++ header files should have the extension *.h*. Source files can have the extension *.c++* (recommended), *.C*, *.cc* or *.cpp*.

```
MyClass.c++, MyClass.h
```

These are all accepted C++ standards for file extension.

2. Classes should be declared in individual header files with the file name matching the class name. Secondary private classes can be declared as inner classes and reside in the file of the class they belong to. All definitions should reside in source files.

```
class MyClass
{
  public:
    int getValue () {return value_;} // NO!
    ...
  private:
    int value_;
}
```

The header files should declare an interface, the source file should implement it. When looking for an implementation, the programmer should always know that it is found in the source file. The obvious exception to this rule is of course inline functions that must be defined in the header file.

3. Special characters like TAB and page break must be avoided. These characters are bound to cause problem for editors, printers, terminal emulators or debuggers when used in a multi-programmer, multi-platform environment.

© Copyright Virtual University of Pakistan

© Copyright Virtual University of Pakistan

Include Files and Include Statements for Java and C++

1. Header files must include a construction that prevents multiple inclusion. The convention is an all uppercase construction of the module name, the file name and the h suffix.

```
#ifndef MOD_FILENAME_H
#define MOD_FILENAME_H
 :
#endif
```

The construction is to avoid compilation errors. The construction should appear in the top of the file (before the file header) so file parsing is aborted immediately and compilation time is reduced.

Classes and Interfaces

Class and Interface declarations should be organized in the following manner:

1. Class/Interface documentation.
2. `class` or `interface` statement.
3. Class (**`static`**) variables in the order **`public`**, **`protected`**, package (no access modifier), **`private`**.
4. Instance variables in the order **`public`**, **`protected`**, package (no access modifier), `private`.
5. Constructors.
6. Methods (no specific order).

© Copyright Virtual University of Pakistan

10.5 Statements in Java and C++

Types

1. Type conversions must always be done explicitly. Never rely on implicit type conversion.

```
floatValue = (float) intValue;     // NOT:    floatValue = intValue;
```

By this, the programmer indicates that he is aware of the different types involved and that the mix is intentional.

2. Types that are local to one file only can be declared inside that file.

3. The parts of a class must be sorted *public*, *protected* and *private*. All sections must be identified explicitly. Not applicable sections should be left out. The ordering is *"most public first"* so people who only wish to use the class can stop reading when they reach the `protected/private` sections.

Variables

1. Variables should be initialized where they are declared and they should be declared in the smallest scope possible.

2. Variables must never have dual meaning. This enhances readability by ensuring all concepts are represented uniquely. Reduce chance of error by side effects.

3. Class variables should never be declared public. The concept of information hiding and encapsulation is violated by public variables. Use private variables and access functions instead. One exception to this rule is when the class is essentially a data structure, with no behavior (equivalent to a C++ `struct`). In this case it is appropriate to make the class' instance variables public.

4. Related variables of the same type can be declared in a common statement. Unrelated variables should not be declared in the same statement.

```
float                 x,                 y,                 z;
float  revenueJanuary, revenueFebrury, revenueMarch;
```

The common requirement of having declarations on separate lines is not useful in the situations like the ones above. It enhances readability to group variables.

5. Variables should be kept alive for as short a time as possible. Keeping the operations on a variable within a small scope, it is easier to control the effects and side effects of the variable.

© Copyright Virtual University of Pakistan

6. Global variables should not be used. Variables should be declared only within scope of their use. Same is recommended for global functions or file scope variables. It is easier to control the effects and side effects of the variables if used in limited scope.

7. Implicit test for *0* should not be used other than for boolean variables and pointers.

```
if (nLines != 0)     // NOT:   if (nLines)
if (value != 0.0)    // NOT:   if (value)
```

It is not necessarily defined by the compiler that ints and floats 0 are implemented as binary 0. Also, by using explicit test the statement give immediate clue of the type being tested. It is common also to suggest that pointers shouldn't test implicit for 0 either, i.e. `if (line == 0)` instead of `if (line)`. The latter is regarded as such a common practice in C/C++ however that it can be used.

Loop structures

1. Only loop control statements must be included in the *for()* construction.

```
sum = 0;                    // NOT:     for (i=0, sum=0; i<100; i++)
for (i=0; i<100; i++)       //             sum += value[i];
  sum += value[i];
```

2. Loop variables should be initialized immediately before the loop.

```
boolean done = false;  // NOT:   boolean  done = false;
while (!done) {        //          :
  :                    //        while (!done) {
}                      //          :
                                  }
```

3. The use of *do .... while* loops should be avoided. There are two reasons for this. First is that the construct is superflous; Any statement that can be written as a *do .... while* loop can equally well be written as a *while* lopp or a *for* loop. Complexity is reduced by minimizing the number of constructs being used. The other reason is of readability. A loop with the conditional part at the end is more difficult to read than one with the conditional at the top.

4. The use of *break* and *continue* in loops should be avoided. These statements should only be used if they prove to give higher readability than their structured counterparts. In general *break* should only be used in *case* statements and *continue* should be avoided alltogether.

5. The form *for (;;)* should be used for empty loops.

```
for (;;) {             // NOT:   while (true) {
  :                    //          :
}                      //        }
```


© Copyright Virtual University of Pakistan

This form is better than the functionally equivalent *while (true)* since this implies a test against *true*, which is neither necessary nor meaningful. The form while(true) should be used for infinite loops.

Conditionals

1. Complex conditional expressions must be avoided. Introduce temporary boolean variables instead.

```
if ((elementNo < 0) || (elementNo > maxElement)||
    elementNo == lastElement) {
  :
}
```

should be replaced by:

```
boolean isFinished      = (elementNo < 0) || (elementNo > maxElement);
boolean isRepeatedEntry = elementNo == lastElement;
if (isFinished || isRepeatedEntry) {
  :
}
```

2. The nominal case should be put in the *if*-part and the exception in the *else*-part of an if statement.

```
boolean isError = readFile (fileName);
if (!isError) {
  :
}
else {
  :
}
```

3. The conditional should be put on a separate line.

```
if (isDone)               // NOT:  if (isDone) doCleanup();
  doCleanup();
```

4. Executable statements in conditionals must be avoided.

```
file = openFile (fileName, "w");  // NOT:   if ((file = openFile
(fileName, "w")) != null) {
if (file != null) {               //          :
  :                               //        }
}
```


© Copyright Virtual University of Pakistan

Miscellaneous

1. The use of magic numbers in the code should be avoided. Numbers other than *0* and *1* should be considered declared as named constants instead.

2. Floating point constants should always be written with decimal point and at least one decimal.

```
double total = 0.0;    // NOT: double total = 0;
double speed = 3.0e8; // NOT: double speed = 3e8;

double sum;
:
sum = (a + b) * 10.0;
```

This emphasizes the different nature of integer and floating point numbers even if their values might happen to be the same in a specific case. Also, as in the last example above, it emphasize the type of the assigned variable (`sum`) at a point in the code where this might not be evident.

3. Floating point constants should always be written with a digit before the decimal point.

```
double total = 0.5;    // NOT: double total = .5;
```

The number and expression system in Java is borrowed from mathematics and one should adhere to mathematical conventions for syntax wherever possible. Also, 0.5 is a lot more readable than .5; There is no way it can be mixed with the integer 5.

4. Functions in C++ must always have the return value explicitly listed.

```
int getValue()      // NOT: getValue()
{
  :
}
```

If not explicitly listed, C++ implies `int` return value for functions.

5. `goto` in C++ should not be used. Goto statements violates the idea of structured code. Only in some very few cases (for instance breaking out of deeply nested structures) should goto be considered, and only if the alternative structured counterpart is proven to be less readable.

© Copyright Virtual University of Pakistan

Lecture No. 30

10.6 Layout and Comments in Java and C++

Comments

The problem with comments is that they lie. Comments are not syntax checked, there is nothing forcing them to be accurate. And so, as the code undergoes change during schedule crunches, the comments become less and less accurate.

As Fowler puts it, comments should not be used as deodorants. Tricky code should not be commented but rewritten. In general, the use of comments should be minimized by making the code self-documenting by appropriate name choices and an explicit logical structure.

If, however, there is a need to write comments for whatever reason, the following guidelines should be observed.

1. All comments should be written in English. In an international environment English is the preferred language.

2. Use // for all comments, including multi-line comments.

```
// Comment spanning
// more than one line
```

Since multilevel commenting is not supported in C++ and Java, using // comments ensure that it is always possible to comment out entire sections of a file using /* */ for debugging purposes etc.

3. Comments should be indented relative to their position in the code.

```
while (true) {          // NOT:    while (true) {
  // Do something       //         // Do something
  something();          //             something();
}                       //         }
```


© Copyright Virtual University of Pakistan

10.7 Expressions and Statements

Layout

1. Basic indentation should be 2.

```
for (i = 0; i < nElements; i++)
  a[i] = 0;
```

Indentation of 1 is to small to emphasize the logical layout of the code. Indentation larger than 4 makes deeply nested code difficult to read and increase the chance that the lines must be split. Choosing between indentation of 2, 3 and 4, 2 and 4 are the more common, and 2 chosen to reduce the chance of splitting code lines.

Natural form for expression

Expression should be written as if they are written as comments or spoken out aloud. Conditional expression with negation are always difficult to understand. As an example consider the following code:

if (! (block < activeBlock) || !(blockId >= unblocks))

The logic becomes much easier to follow if the code is written in the natural form as shown below:

if ((block >= activeBlock) || (blockId < unblocks))

Parenthesize to remove ambiguity

Parentheses should always be used as they reduce complexity and clarify things by specifying grouping. It is especially important to use parentheses when different unrelated operators are used in the same expression as the precedence rules are often assumed by the programmers, resulting in logical errors that are very difficult to spot. As an example consider the following statement:

if (x & MASK == BITS)

This causes problems because == operator has higher precedence than & operator. Hence, MASK and BITS are first compared for equality and then the result, which is 0 or 1, is andded with x. This kind of error will be extremely hard to catch. If, however, parentheses are used, there will be no ambiguity as shown below.

if ((x & MASK) == BITS)

© Copyright Virtual University of Pakistan

Following is another example of the use of parentheses which makes the code easier to understand and hence easier to maintain.

```
leapYear = year % 4 == 0  && year % 100 != 0 || year % 400 == 0 ;
```

In this case parentheses have not been used and therefore the definition of a leap year is not very clear for the code. The code becomes self explanatory with the help of proper use of parentheses as shown below:

```
leapYear = ((year % 4 == 0)  && (year % 100 != 0))  ||
                    (year % 400 == 0);
```

Breakup complex expressions

Complex expressions should be broken down into multiple statements. An expression is considered to be complex if it uses many operators in a single statement. As an example consider the following statement:

```
*x += (*xp=(2*k < (n-m) ? c[k+1] : d[k--]));
```

This statement liberally uses a number of operators and hence is very difficult to follow and understand. If it is broken down into simple set of statements, the logic becomes easier to follow as shown below:

```
if (2*k < n-m)
        *xp = c[k+1];
else
        *xp = d[k--];
*x = *x + *xp;
```

10.8 Shortcuts and cryptic code

Sometimes the programmers, in their creative excitement, try to write very concise code by using shortcuts and playing certain kinds of tricks. This results in a code which is cryptic in nature and hence is difficult to follow. Maintenance of such code therefore becomes a nightmare. Following are some examples of such code.

1. Let us start with a very simple shortcut, often used by programmers. Assume that we have the following statement.

   ```
   x *= a;
   ```

   For some reason the code was later modified to:

© Copyright Virtual University of Pakistan

```
x *= a + b;
```

This seemingly harmless change is actually a little cryptic and causes confusion. The problem lies with the semantics of this statement. Does it mean x = x*a+b or x = x*(a+b)? The second one is the right answer but is not obvious from the syntax and hence causes problems.

2. Let us now look at a more complex example. What is the following code doing?

```
subkey = subkey >> (bitoff – (bitoff >> 3) << 3));
```

As can be seen, it is pretty hard to understand and therefore difficult to debug in case there is any problem. What this code is actually doing is masking bitoff with octal 7 and then use the result to shift subkey those many time. This can be written as follows:

```
subkey = subkey >> (bitoff & 0x7);
```

It is quite evident that the second piece of code is much follow to read than the first one.

3. The following piece of code is taken from a commercial software:

```
a = a >> 2;
```

It is easy to see that a is shifted right two times. However, the real semantics of this code are hidden – the real intent here is to divide a by 4. No doubt that the code above achieves this objective but it is hard for the reader to understand the intent as to why a is being shifted right twice. It would have been much better had the code been written as follows:

```
a = a/4;
```

4. A piece of code similar to the following can be found in many data structures books and is part of circular implementation of queues using arrays.

```
bool Queue::add(int n)
{
        int k = (rear+1) % MAX_SIZE;
        if (front == k)
                return false;
        else {
                rear = k;
                queue[rear] = n;
                return true;
        }
}
```


© Copyright Virtual University of Pakistan

```
bool Queue::isFull()
{
        if (front == (rear+1 % MAX_SIZE))
                return true;
        else
                return false;
}
```

This code uses the % operator to set the rear pointer to 0 once it has reached MAX_SIZE. This is not obvious immediately. Similarly, the check for queue full is also not trivial.

In an experiment, the students were asked to implement double-ended queue in which pointer could move in both directions. Almost everyone made the mistake of writing something like rear = (rear-1) % MAX_SIZE. This is because the semantics of % operation are not obvious.

It is always much better to state the logic explicitly. Also, counting is also much easier to understand and code as compared to some tricky comparisons (e.g. check for isFull in this case). Application of both these principles resulted in the following code. It is easy to see that this code is easier to understand. It is interesting to note that when another group of students were asked to do implement double-ended queue after showing them this code, almost everyone did it without any problems.

```
bool Queue::add()
{
        if (! isFull() ) {
                rear++;
                if (rear == MAX_SIZE) rear = 0;
                QueueArray[rear] = n;
                size++;
                return true;
        }
        else return false;
}

bool Queue::isFull(int n)
{
        if (size == MAX_SIZE)
                return true;
        else
                return false;
}
```


© Copyright Virtual University of Pakistan

Lecture No. 31

Coding Style Guidelines (Continued)

Switch Statement

In the switch statement, cases should always end with a break. A tricky sequence of fall-through code like the one below causes more trouble than being helpful.

```
switch(c) {
case '-'  : sign = -1;
case '+'  : c = getchar();
case '.'  : break;
default   : if (! isdigit(c))
                        return 0;
}
```

This code is cryptic and difficult to read. It is much better to explicitly write what is happening, even at the cost of duplication.

```
switch(c) {
case '-':   sign = -1;
                    c = getchar();
                    break;
case '+':   c = getchar();
                    break;
case '.':   break;
default:    if (! isdigit(c))
                        return 0;
                    break;
}
```

It would even be better if such code is written using the if statement as shown below.

```
if (c == '-') {
        sign = -1;
        c = getchar();
}
else if (c == '+') {
        c = getchar();
}
else if (c != '.' && !isdigit(c)) {
        return 0;
}
```


© Copyright Virtual University of Pakistan

Magic Numbers

Consider the following code segment:

```
fac = lim / 20;
if (fac < 1)
        fac = 1;
for (i =0, col = 0; i < 27; i++, j++) {
        col += 3;
        k = 21 – (let[i] /fac);
        star = (let[i] == 0) ? ‘ ’ : ‘*’;
        for (j = k; j < 22; j++)
                draw(j, col, star);
}
draw(23, 1, ‘ ’);
for (i=‘A’; i <= ‘Z’; i++)
        cout << i;
```

Can you tell by reading the code what is meant by the numbers 20, 27, 3, 21, 22, and 23. These are constant that mean something but they do not give any indication of their importance or derivation, making the program hard to understand and modify. To a reader they work like magic and hence are called magic numbers. Any number (even 0 or 1) used in the code is a magic number. It should rather have a name of its own that can be used in the program instead of the number.

The difference would be evident if we look at the code segment below that achieves the same purpose as the code above.

```
enum {
        MINROW = 1,
        MINCOL = 1,
        MAXROW = 24,
        MAXCOL = 80,
        LABELROW = 1,
        NLET = 26,
        HEIGHT = MAXROW –4,
        WIDTH = (MAXCOL-1) / NLET
};

fac = (lim+HEIGHT-1) /HEIGHT;
if (fac < 1)
        fac = 1;
for (i =0; i < NLET; i++) {
        if (let[i] == 0)
                continue;
        for (j = HEIGHT – let[i] / fac; j < HEIGHT; j++)
                draw(j-1 + LABELROW,
                        (i+1)*WIDTH, ‘*’);
}

draw(MAXROW-1, MINCOL+1, ‘ ’);
```


© Copyright Virtual University of Pakistan

```
for (i='A'; i <= 'Z'; i++)
        cout << i;
```

Use (or abuse) of Zero

The number 0 is the most abused symbol in programs written in C or C++. One can easily find code segment that 0 in a fashion similar to the examples below in almost every C/C++ program.

```
flag = 0;          // flag is boolean
str = 0;           // str is string
name[i] = 0;       // name is char array
x = 0;             // x is floating pt
i = 0;             // i is integer
```

This is a legacy of old style C programming. It is much better to use symbols to explicitly indicate the intent of the statement. It is easy to see that the following code is more in line with the self-documentation philosophy than the code above.

```
flag = false;
str = NULL;
name[i] = '\0';
x = 0.0;
i = 0;
```


© Copyright Virtual University of Pakistan

Lecture No. 32

10.9 Clarity through modularity

As mentioned earlier, abstraction and encapsulation are two important tools that can help in managing and mastering the complexity of a program. We also discussed that the size of individual functions plays a significant role in making the program easy or difficult to understand. In general, as the function becomes longer in size, it becomes more difficult to understand. Modularity is a tool that can help us in reducing the size of individual functions, making them more readable. As an example, consider the following selection sort function.

```
void selectionSort(int a[], int size)
{
        int i, j;
        int temp;
        int min;

        for (i = 0; i < size-1; i++){
                min = i;
                for (j = i+1; j < size; j++){
                        if (a[j] < a[min])
                                min = j;
                }
                temp = a[i];
                a[i] = a[min];
                a[min] = temp;
        }
}
```

Although it is not very long but we can still improve its readability by breaking it into small functions to perform the logical steps. The modified code is written below:

© Copyright Virtual University of Pakistan

```
void swap(int &x, int &y)
{
        int temp;
        temp = x;
        x = y;
        y = temp;
}

int minimum(int a[], int from, int to)
{
        int i;
        int min;
        min = a[from];
        for (i = from; i <= to; i++){
                if (a[i] < a[min])
                        min = i;
        }
        return min;
}

void selectionSort(int a[], int size)
{
        int min;
        int i;
        for (i = 0; i < size; i++){
                min = minimum(a, i, size –1);
                swap(a[i], a[min]);
        }
}
```

It is easy to see that the new selectionSort function is much more readable. The logical steps have been abstracted out into the two functions namely, minimum and swap. This code is not only shorter but also as a by product we now have two functions (minimum and swap) that can be reused.

Reusability is one of the prime reasons to make functions but is not the only reason. Modularity is of equal concern (if not more) and a function should be broken into smaller pieces, even if those pieces are not reused. As an example, let us consider the quickSort algorithm below.

© Copyright Virtual University of Pakistan

```
void quickSort(int a[], int left, int right)
{
        int i, j;
        int pivot;
        int temp;
        int mid = (left + right)/2;
        if (left < right){
                i = left - 1;
                j = right + 1;
                pivot = a[mid];
                do {
                        do i++; while (a[i] < pivot);
                        do j--; while (a[i] < pivot);
                        if (i<j){
                                temp = a[i];
                                a[i] = a[j];
                                a[j] = temp;
                        }
                } while (i < j);
                temp = a[left];
                a[left] = a[j];
                a[j] = temp;

                quickSort(a, left, j);
                quickSort(a, j+1, right);
}
```

This is actually a very simple algorithm but students find it very difficult to remember. If is broken in logical steps as shown below, it becomes trivial.

```
void quickSort(int a[], int left, int right)
{
        int p;
        if (left < right){
                p = partition(a, left, right);
                quickSort(a, left, p-1);
                quickSort(a, p+1, right);
        }
}


int partition(int a[], int left, int right)
{
        int i; j;
        int pivot;
        i = left + 1;
        j = right;
        pivot = a[left];
        while(i < right && a[i] < pivot) i++;
        while(j > left && a[j] >= pivot) j++;
        if(i < j)
                swap(a[i], a[j]);
        swap(a[left], a[j]);
        return j;
}
```


© Copyright Virtual University of Pakistan

10.10 **Short circuiting || and &&**

The logical and operator, &&, and logical or operators, ||, are special due to the C/C++ short circuiting rule, i.e. a || b and a && b are short circuit evaluated. That is, logical expressions are evaluated left to right and evaluation stops as soon as the final truth value can be determined.

Short-circuiting is a very useful tool. It can be used where one boolean expression can be placed first to "guard" a potentially unsafe operation in a second boolean expression. Also, time is saved in evaluation of complex expressions using operators || and &&. However, a number of issues arise if proper attention is not paid.

Let us look at the following code segment taken from a commercially developed software for a large international bank:

```
struct Node {
        int data;
        Node * next;
};


Node *ptr;
...

while (ptr->data < myData && ptr != NULL){
        // do something here
}
```

What's wrong with this code?

The second part of condition, ptr != NULL, is supposed to be the guard. That is, if the value of the pointer is NULL, then the control should not enter the body of the while loop otherwise, it should check whether ptr->data < myData or not and then proceed accordingly. When the guard is misplaced, if the pointer is NULL then the program will crash because it is illegal to access a component of a non-existent object. This code is rewritten as follows. This time the short-circuiting helps in achieving the desired objective which would have been a little difficult to code without such help.

```
while (ptr != NULL && ptr->data < myData){
        // do something here
}
```


© Copyright Virtual University of Pakistan

10.11 Operand Evaluation Order and Side Effects

A side effect of a function occurs when the function, besides returning a value, changes either one of its parameters or a variable declared outside the function but is accessible to it. That is, a side effect is caused by an operation that may return an explicit result but it may also modify the values stored in other data objects. Side effects are a major source of programming errors and they make things difficult during maintenance or debugging activities. Many languages do not specify the function evaluation order in a single statement. This combined with side effects causes major problems. As an example, consider the following statement:

```
c = f1(a) + f2(b);
```

The question is, which function (f1 or f2) will be evaluated first as the C/C++ language does not specify the evaluation order and the implementer (compiler writer) is free to choose one order or the other. The question is: does it matter?

To understand this, let's look at the definition of f1 and f2.

```
int f1(int &x)
{
        x = x * 2;
        return x + 1;
}

int f2(int &y)
{
        y = y / 2;
        return y - 1;
}
```

In this case both f1 and f2 have side effects as they both are doing two things - changing the value of the parameter and changing the value at the caller side. Now if we have the following code segment,

```
a = 3;
b = 4;

c = f1(a) + f2(b);
```

then the value of a, b, and c would be as follows:

```
a = 6
b = 2
c = 8
```


© Copyright Virtual University of Pakistan

So far there seem to be any problem. But let us now consider the following statement:

```
c = f1(a) + f2(a);
```

What will be the value of a and c after this statement?

If f1 is evaluated before f2 then we have the following values:

```
a = 3
c = 9 // 7 + 2
```

On the other hand, if f2 is evaluated before f1 then, we get totally different results.

```
a = 2
c = 3 // 3 + 0
```


© Copyright Virtual University of Pakistan

Lecture No. 33

Common Coding mistakes

Following is short list of common mistakes made due to side-effects.

1.
```
array[i++] = i;
```
If i is initially 3, array[3] might be set to 3 or 4.

2.
```
array[i++] = array[i++] = x;
```
Due to side effects, multiple assignments become very dangerous. In this example, a whole depends upon when i is incremented.

3.
“,” is very dangerous as it causes side effects. Let’s look at the following statement:

```
int i, j = 0;
```

Because of the syntax, many people would assume that i is also being initialized to 0, while it is not. Combination of , and = -- is fatal. Look at the following statement:

```
a = b, c = 0;
```

A majority of the programmers would assume that all a, b, and c are being initialized to 0 while only c is initialized and a and b have garbage values in them. This kind of overlook causes major programming errors which are not caught easily and are caused only because there are side effects.

Guidelines

If the following guidelines are observed, one can avoid hazards caused by side effects.

1. never use “,” except for declaration
2. if you are initializing a variable at the time of declaration, do not declare another variable in the same statement
3. never use multiple assignments in the same statement
4. Be very careful when you use functions with side effects – functions that change the values of the parameters.
5. Try to avoid functions that change the value of some parameters and return some value at the same time.

© Copyright Virtual University of Pakistan

10.12 Performance

In many cases, performance and maintainability are at odds with one another. When planning for performance, one should always remember the 80/20 rule - you spend 80 percent of your time in 20 percent of the code. That is, we should not try to optimize everything. The proper approach is to profile the program and then identify bottlenecks to be optimized. This is similar to what we do in databases – we usually normalize the database to remove redundancies but then partially de-normalize if there are performance issues.

As an example, consider the following. In this example a function isspam is profiled by calling 10000 times. The results are shown in the following table:

sec	%	Cum%	cycles	instructions	calls	function
			990000	10000	00	strchr
			80000	6000	00	strncmp
			0000	0000	00	strstr
			5559	2213	35	strlen
			000	000		isspam
11 other functions with insignificant performance overhead						

The profiling revealed that most of time was spent in strchr and strncmp and both of these were called from strstr.

When a small set (a couple of functions) of functions which use each other is so overwhelmingly the bottleneck, there are two alternatives:

1. use a better algorithm
2. rewrite the whole set

In this particular case strstr was rewritten and profiled again. It was and found out that although it was much faster but now 99.8% of the time was spent in strstr.

The algorithm was rewritten and restructured again by eliminating strstr, strchr, and strncmp and used memcmp. Now memcmp was much more complex than strstr but it gained efficiency by eliminating a number of loops and the new results are as shown:

sec	%	Cum%	cycles	instructions	calls	function
			0000	90000	000	memcmp
			0000	0000		isspam
			4	3		strlen

strlen now went from over two million calls to 652.

© Copyright Virtual University of Pakistan

Many details of the execution can be discovered by examining the numbers. The trick is to concentrate on hot spots by first identifying them and then cooling them. As mentioned earlier, most of the time is spent in loops. Therefore we need to concentrate on loops.

As an example, consider the following:

```
for (j = i; j < MAX_FIELD; j++)
  clear(j);
```

This loop clears field before each new input is read. It was observed that it was taking almost 50% of the total time. On further investigation it was found out that MAX_FIELD was 200 but the actual fields that needed to be cleared were 2 or 3 in most cases. The code was subsequently modified as shown below:

```
for (j = i; j < maxField; j++)
  clear(j);
```

This reduced the overall execution time by half.

© Copyright Virtual University of Pakistan

Lecture No. 34

10.13 Portability

Many applications need to be ported on to many different platforms. As we have seen, it is pretty hard to write error free, efficient, and maintainable software. So, if a major rework is required to port a program written for one environment to another, it will be probably not come at a low cost. So, we ought to find ways and means by which we can port applications to other platforms with minimum effort. The key to this lies in how we write our program. If we are careful during writing code, we can make it portable. On the other hand if we write code without portability in mind, we may end-up with a code that is extremely hard to port to other environment. Following is brief guideline that can help you in writing portable code.

Stick to the standard

1. Use ANSI/ISO standard C++
2. Instead of using vendor specific language extensions, use STL as much as possible

Program in the mainstream

Although C++ standard does not require function prototypes, one should always write them.

```
double sqrt();          // old style acceptable by ANSI C

double sqrt(double);    // ANSI – the right approach
```

Size of data types

Sizes of data types cause major portability issues as they vary from one machine to the other so one should be careful with them.

```
int i, j, k;

…
j = 20000;
k = 30000;

i = j + k;
```


© Copyright Virtual University of Pakistan

```
                    // works if int is 4 bytes
// what will happen if int is 2 bytes?
```


© Copyright Virtual University of Pakistan

Order of Evaluation

As mentioned earlier during the discussion of side effects, order of evaluation varies from one implementation to other. This therefore also causes portability issues. We should therefore follow guidelines mentioned in the side effect discussion.

Signedness of char

The language does not specify whether char is signed or unsigned.

```
char c;
        // between 0 and 255 if unsigned
        // -128 to 127 if signed

c = getchar();
if (c == EOF)  ??

        // will fail if it is unsigned
```

It should therefore be written as follows:

```
int c;
c = getchar();
if (c == EOF)
```

Arithmetic or Logical Shift

The C/C++ language has not specified whether right shift >> is arithmetic or logical. In the arithmetic shift sign bit is copied while the logical shift fills the vacated bits with 0. This obviously reduces portability.

Interestingly, Java has introduced a new operator to handle this issue. >> is used for for arithmetic shift and >>> for logical shift.

Byte Order and Data Exchange

The order in which bytes of one word are stored is hardware dependent. For example in Intel architecture the lowest byte is the most significant byte while in Motorola architecture the highest byte of a word is the most significant one. This causes problem when dealing with binary data and we need to be careful while exchanging data between to heterogeneous machines. One should therefore only use text for data exchange. One should also be aware of the internationalization issues and hence should not assume ASCII as well as English.

© Copyright Virtual University of Pakistan

Alignment

The C/C++ language does not define the alignment of items within structures, classes, an unions. data may be aligned on word or byte boundaries. For example:

```
struct X {
        char c;
        int i;
};
```

address of i could be 2, 4, or 8 from the beginning of the structure. Therefore, using pointers and then typecasting them to access individual components will cause all sorts of problems.

© Copyright Virtual University of Pakistan

Bit Fields

Bit fields allow the packing of data in a structure. This is especially useful when memory or data storage is at a premium. Typical examples:

- Packing several objects into a machine word. *e.g.* 1 bit flags can be compacted -- Symbol tables in compilers.
- Reading external file formats -- non-standard file formats could be read in. *E.g.* 9 bit integers.

C lets us do this in a structure definition by putting :*bit length* after the variable. *i.e.*

```
struct packed_struct {
        unsigned int f1:1;
        unsigned int f2:1;
        unsigned int f3:1;
        unsigned int f4:1;
        unsigned int type:4;
        unsigned int funny_int:9;
} pack;
```

Here the `packed_struct contains 6 members: Four 1 bit` *flags* `f1..f3, a 4 bit type and a 9 bit funny_int.`

C automatically packs the above bit fields as compactly as possible, provided that the maximum length of the field is less than or equal to the integer word length of the computer. If this is not the case then some compilers may allow memory overlap for the fields whilst other would store the next field in the next word.

Bit fields are a convenient way to express many difficult operations. However, bit fields do suffer from a lack of portability between platforms:

- integers may be signed or unsigned
- Many compilers limit the maximum number of bits in the bit field to the size of an `integer` which may be either 16-bit or 32-bit varieties.
- Some bit field members are stored left to right others are stored right to left in memory.
- If bit fields too large, next bit field may be stored consecutively in memory (overlapping the boundary between memory locations) or in the next word of memory.

Bit fields therefore should not be used.

© Copyright Virtual University of Pakistan

Lecture No. 35

10.14 Exception handling

Exception handling is a powerful technique that separates error-handling code from normal code. It also provides a consistent error handling mechanism. The greatest advantage of exception handling is its ability to handle asynchronous errors.

The idea is to raise some error flag every time something goes wrong. There is a system that is always on the lookout for this error flag. Third, the previous system calls the error handling code if the error flag has been spotted. The raising of the imaginary error flag is simply called *raising* or *throwing* an error. When an error is thrown the overall system responds by *catching* the error. Surrounding a block of error-sensitive code with exception handling is called *trying* to execute a block. The following code segment illustrates the general exception handling mechanism.

```
try {
___...
___...
___throw Exception()
___...
___...
} catch( Exception e )
{
___...
___...
}
```

One of the most powerful features of exception handling is that an error can be thrown over function boundaries. This allows programmers to put the error handling code in one place, such as the *main*-function of your program.

© Copyright Virtual University of Pakistan

Exceptions and code complexity

A number of invisible execution paths can exist in simple code in a language that allows exceptions. The complexity of a program may increase significantly if there are exceptional paths in it. Consider the following code:

```
String EvaluateSalaryAnadReturnName( Employee e)
{
  if (e.Title() == "CEO" || e.Salary() > 10000)
  {
        cout << e.First() << " " << e.Last() << " is overpaid" << endl;
  }
  return e.First() + " " + e.Last();
}
```

Before moving any further, let's take the following assumptions:

1. Different order of evaluating function parameters are ignored.
2. Failed destructors are ignored
3. Called functions are considered atomic
    a. for example: e.Title() could throw for several reasons but all that matters for this function is whether e.Title() results in an exception or not.
4. To count as different execution paths, an execution path must be made-up of a unique sequence of function calls performed and exited in the same way.

Question: How many more execution paths are there?

Ans: 23. There are 3 non-exceptional paths and 20 exceptional paths.

The non-exceptional paths:

The three non-exceptional paths are enumerated as below:

1. if (e.Title() == "CEO" || e.Salary() > 10000)
    - if e.Title() == "CEO" is true then the second part is not evaluated and e.Salary() will not be called.
    - cout will be performed

2. if e.Title() != "CEO" and e.Salary() > 10000
    - both parts of the condition will be evaluated
    - cout will be performed.

3. if e.Title() != "CEO" and e.Salary() <= 10000
    - both parts of the condition will be evaluated
    - cout will not be performed.

© Copyright Virtual University of Pakistan

Exceptional Code Paths

The 20 exceptional code path are listed below.

1. String EvaluateSalaryAnadReturnName(Employee e)

   The argument is passed by value, which invokes the copy constructor. This copy operation might throw an exception.

2. if (e.Title() == “CEO” || e.Salary() > 10000)

   e.Title() might itself throw, or it might return an object of class type by value, and that copy operation might throw.

3. if (e.Title() == “CEO” || e.Salary() > 10000)

   Same as above.

4. if (e.Title() == “CEO” || e.Salary() > 10000)

   To match a valid ==() operator, the string literal may need to be converted to a temporary object of class type and that construction of the temporary might throw.

5. if (e.Title() == “CEO” || e.Salary() > 10000)

   Same as above.

6. if (e.Title() == “CEO” || e.Salary() > 10000)

   operator ==() might throw.

7. if (e.Title() == “CEO” || e.Salary() > 10000)

   Same as above.

© Copyright Virtual University of Pakistan

8. if (e.Title() == “CEO” || e.Salary() > 10000)

Same as above.

9-13 cout << e.First() << “ “ << e.Last() << “ is overpaid” << endl;

As per C++ standard, any of the five calls to << operator might throw.

14-15 cout << e.First() << “ “ << e.Last() << “ is overpaid” << endl

similar to 2 and 3.

16-17 return e.First() + “ “ + e.Last();

similar to 14-15.

18-19 return e.First() + “ “ + e.Last();

similar to 6,7, and 8.

20. return e.First() + “ “ + e.Last();

similar to 4.

Summary:

- A number of invisible execution paths can exist in simple code in a language that allows exceptions.
- Always be exception-aware. Know what code might emit exceptions.

© Copyright Virtual University of Pakistan

The Challenge:
Can we make this code exception safe and exception neutral? That is, rewrite it (if needed) so that it works properly in the presence of an exception and propagates all exceptions to the caller?

Exception-Safety:
A function is exception safe if it might throw but do not have any side effects if it does throw and any objects being used, including temporaries, are exception safe and clean-up there resources when destroyed.

Exception Neutral:
A function is said to be exception neutral if it propagates all exceptions to the caller.

Levels of Exception Safety
- Basic Guarantee: Ensures that temporaries are destroyed properly and there are no memory leaks.
- Strong Guarantee: Ensures basic guarantee as well as there is full-commit or roll-back.
- No-throw Guarantee: Ensure that a function will not throw.

Exception-safety requires either no-throw guarantee or basic and strong guarantee.

Does the function satisfy basic guarantee?
Yes. Since the function does not create any objects, in the presence of an exception, it does not leak any resources.

Does the function satisfy strong guarantee?
No. The strong guarantee says that if the function fails because of an exception, the program state must not be change.

This function has two distinct side-effects:
- an overpaid message is emitted to cout.
- A name strings is returned.

As far as the second side-effect is concerned, the function meets the strong guarantee because if an exception occurs the value will never be returned.

As far as the first side-effect is concerned, the function is not exception safe for two reasons:
- if exception is thrown after the first part of the message has been emitted to cout but before the message has been completed (for example if the fourth << operator throws), then a partial message was emitted to cout.
- If the message emitted successfully but an exception occurs in later in the function (for example during the assembly of the return value), then a

© Copyright Virtual University of Pakistan

message was emitted to cout even though the function failed because of an exception. It should be complete commit or complete roll-back.

Does the function satisfy no-throw guarantee?

No. This is clearly not true as lots of operations in the function might throw.

Strong Guarantee

To meet strong guarantee, either both side-effects are completed or an exception is thrown and neither effect is performed.

```
// First attempt:

String EvaluateSalaryAnadReturnName( Employee e)
{
        String result = e.First() + " " + e.Last();

        if (e.Title() == "CEO" || e.Salary() > 10000)
        {

                String message = result + " is overpaid\n";
                cout << message;

        }
        return result;
}
```

What happens if the function is called as follows:

```
String theName;
theName = evaluateSalarayAndReturnName(someEmplyee);
```

1. string copy constructor is invoked because the result is returned by value.
2. The copy assignment operator is invoked to copy the result into theName
3. If either copy fails then the function has completed its side-effects (since the message was completely emitted and the return value was completely constructed) but the result has been irretrievable lost.

© Copyright Virtual University of Pakistan

Can we do better and perhaps avoid the problem by avoiding the copy?

```
// Second attempt:

String EvaluateSalaryAnadReturnName( Employee e, String &r)
{
        String result = e.First() + " " + e.Last();

        if (e.Title() == "CEO" || e.Salary() > 10000)
        {

                String message = result + " is overpaid\n";
                cout << message;

        }
        r = result;
}
```

Looks better but assignment to r might still fail which leaves us with one side-effect completed and other incomplete.

```
// Third attempt:

auto_ptr<String> EvaluateSalaryAnadReturnName( Employee e)
{
        auto_ptr<String> result = new String(e.First() + " " + e.Last());

        if (e.Title() == "CEO" || e.Salary() > 10000)
        {

                String message = (*result) + " is overpaid\n";
                cout << message;

        }
        return result;();       // rely on transfer of ownership
                                // this can't throw
}
```

We have effectively hidden all the work to construct the second side-effect (the return value), while we ensured that it can be safely returned to the caller using only non-throwing operation after the first side-effect has completed the printing of the message. In this case we know that once the function is complete, the return value will make successfully into the hands of the caller and be correctly cleaned-up in all cases. This is because the aut_ptr semantics guarantee that If the caller accepts the returned value, the act of accepting a copy of the auto_ptr causes the caller to take the ownership and if the caller does not accept the returned value, say by ignoring the

© Copyright Virtual University of Pakistan

return value, the allocated string will automatically be destroyed with proper clean-up.

Exception Safety and Multiple Side-effects

It is difficult and some-times impossible to provide strong exception safety when there are two or more side-effects in one function and these side-effects are not related with each other. For example we would not have been able to provide exception safety if there are two output messages, one to cout and the other one to cerr. This is because the two cannot be combined.

When such a situation comes with two or more unrelated side-effects which cannot be combined then the best way to handle such a situation is break it into two separate functions. That way, at least, the caller would know that these are two separate atomic steps.

Summary

1. Providing the strong exception-safety guarantee often requires you to trade-off performance.
2. If a function has multiple un-related side-effects, it cannot always be made strongly exception safe. If not, it can be done only by splitting the function into several functions, each of whose side-effects can be performed atomically.
3. Not all functions need to be strongly exception-safe. Both the original code and attempt#1 satisfy the basic guarantee. For many clients, attempt # 1 is sufficient and minimizes the opportunity for side-effects to occur in the exceptional situation, without requiring the performance trade-off of attempt #3.

© Copyright Virtual University of Pakistan

Lecture No. 36

Software Verification and Validation

11.1 Software Testing

To understand the concept of software testing correctly, we need to understand a few related concepts.

Software verification and validation

Verification and validation are the processes in which we check a product against its specifications and the expectations of the users who will be using it. According to a known software engineering expert Berry Boehm, verification and validation are

Verification

- Does the product meet system specifications?
- Have you built the product right?

Validation

- Does the product meet user expectations?
- Have you built the right product?

It is possible that a software application may fulfill its specifications but it may deviate from users expectations or their desired behavior. That means, software is verified but not validated. How is it possible? It is possible because during the requirements engineering phase, user needs might not have been captured precisely or the analyst might have missed a major stakeholder in the analysis. Therefore, it is important to verify as well as validate the software product.

11.2 Defect

The second major and a very important concept is Defect. A defect is a variance from a desired product attribute. These attributes may involve system specifications well as user expectation. Anything that may cause customer dissatisfaction, is a defect. Whether these defects are in system specifications or in the software products, it is essential to point these out and fix.

Therefore software defect is that phenomenon in which software deviates from its expected behavior. This is non-compliance from the expected behavior with respect to written specifications or the stakeholder needs.

Software and Defect

Software and defects go side by side in the software development life cycle. According to a famous saying by Haliburton, Death and taxes are inevitable. According to Kernighan: Death, taxes, and bugs are the only certainties in the life of a programmer. Software and defects cannot be separated, however, it is important to learn how discovering defects at an appropriate stage improves the software quality. Therefore, software application needs to be verified as well as validated for a successful deployment.

Software Testing

With these concepts, we are in a position to define software testing. Software testing is the process of examining the software product against its requirements. Thus it is a process that involves verification of product with respect to its written requirements and conformance of requirements with user needs. From another perspective, software testing

© Copyright Virtual University of Pakistan

is the process of executing software product on test data and examining its output vis-à-vis the documented behavior.

Software testing objective

- The correct approach to testing a scientific theory is not to try to verify it, but to seek to refute the theory. That is to prove that it has errors. (Popper 1965)
- The goal of testing is to expose latent defects in a software system before it is put to use.
- A software tester tries to break the system. The objective is to show the presence of a defect not the absence of it.
- Testing cannot show the absence of a defect. It only increases your confidence in the software.
- This is because exhaustive testing of software is not possible – it is simply too expansive and needs virtually infinite resources.

Successful Test

From the following sayings, a successful test can be defined
"If you think your task is to find problems then you will look harder for them than if you think your task is to verify that the program has none" – Myers 1979.

"A test is said to be successful if it discovers an error" – doctor's analogy.

The success of a test depends upon the ability to discover a bug not in the ability to prove that the software does not have one. As, it is impossible to check all the different scenarios of a software application, however, we can apply techniques that can discover potential bugs from the application. Thus a test that helps in discovering a bug is a successful test. In software testing phase, our emphasis is on discovering all the major bugs that can be identified by running certain test scenarios. However it is important to keep in mind that testing activity has certain limitations.

Limitations of testing

With the help of the following example, we shall see how difficult it may become to discover a defect from a software application.

Example (adapted from Backhouse)

This is a function that compares two strings of characters stored in an array for equality. The tester who has to test this function has devised the following test cases in which different combinations of inputs are tried and their expected behavior is documented. From the look of this table, it seems that almost all major combinations of inputs have been documented.

Inputs and Expected Outputs

A	b	Expected result
"cat"	"dog"	False
""	""	True
"hen"	"hen"	True
"hen"	"heN"	False
" "	""	False

© Copyright Virtual University of Pakistan

“”	“ball”	False
“cat”	“”	False
“HEN”	“hen”	False
“rat”	“door”	False
“ ”	“ ”	True

Results of testing

The tester runs all the above-mentioned test cases and function returns the same results as expected. But it is still not correct. What can be a problem To analyze the problem, lets look at the code of this string equal routine

Code of the Function

```
bool isStringsEqual(char a[], char b[])
{
   bool result;
   if (strlen(a) != strlen(b))
      result = false;
   else {
      for (int i =0; i < strlen(a); i++)
         if (a[i] == b[i])
            result = true;
         else result = false;
   }
   return result;
}
```

Analysis of the code

It passes all the designated tests but fails for two different strings of same length ending with the same character. For example, “cut” and “rat” would results in true which is not correct.

The above-mentioned defect signifies a clear limitation of the testing process in discovering a defect which is not very frequent. However, it should be noted from this example that a tester cannot generate all possible combinations of test cases to test an application as the number of scenarios may become exhaustive.

Testing limitations

- In order to prove that a formula or hypothesis is incorrect all you have to do to show only one example in which you prove that the formula or theorem is not working.
- On the other hand, million of examples can be developed to support the hypothesis but this will not prove that it is correct.
- These examples only help you in coming up with a hypothesis but they are not proves by themselves and they only enhance your comfort level in that particular hypothesis or in this particular case, in your piece of software.
- You cannot test a program completely because:
  - the domain of the possible inputs is too large to test.
  - there are too many possible paths through the program to test.
- According to Discrete Mathematics

© Copyright Virtual University of Pakistan

- To prove that a formula or hypothesis is incorrect you have to show only one example.
- To prove that it is correct any numbers of examples are insufficient. You have to give a formal proof of its correctness.

11.3 Test Cases and Test Data

In order to test a software application, it is necessary to generate test cases and test data which is used in the application. Test cases correspond to application functionality such that the tester writes down steps which should be followed to achieve certain functionality. Thus a test case involves

- Input and output specification plus a statement of the function under test.
- Steps to perform the function
- Expected results that the software application produces

However, test data includes inputs that have been devised to test the system.

Lecture No. 37

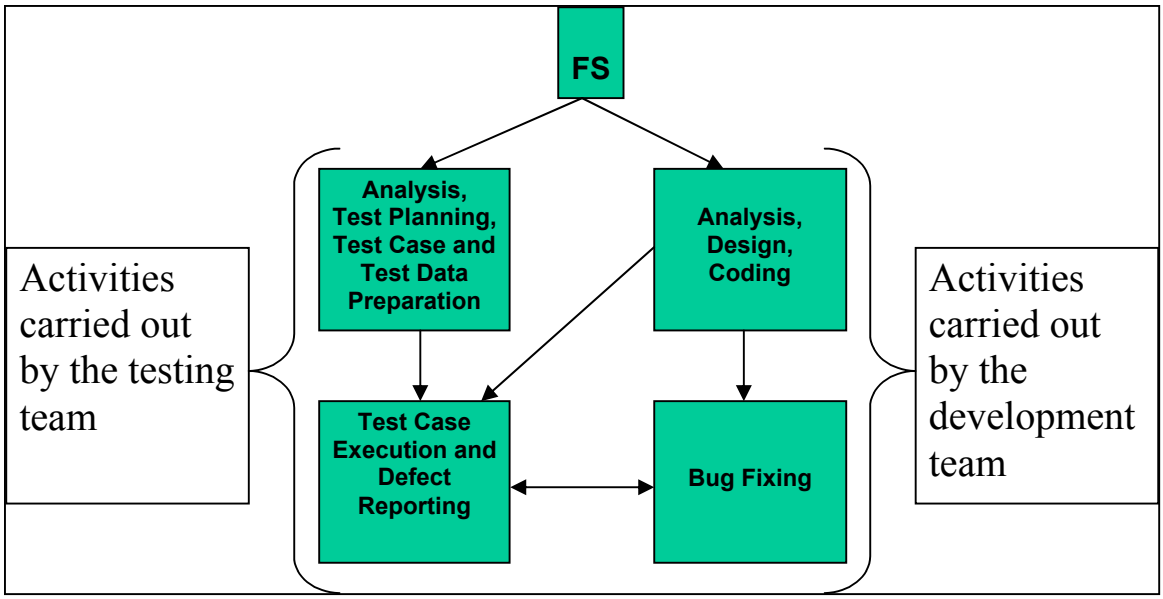
11.3 Testing vs. development

Testing is an intellectually demanding activity and has a lifecycle parallel to software development. A common misperception about testing is that it is not a challenging activity. It should be noted here that the testing demands grip over the domain and application functionality as is required from an analyst or a designer who has to develop the application from requirements. As without having an in-depth knowledge about the system and the requirements from users, a tester cannot write test cases that can verify and validate software application with respect to documented specifications and user needs. Writing test cases and generating test data are processes that demand scenario-building capabilities. These activities essentially require destructive instincts in a tester for the purpose of breaking system to discover loopholes into its functionality.

At the time when these two activities are being performed, merely initial design of the application is completed. Therefore, tester uses his/her imagination to come up with use patterns of the application that can help him/her in describing exact steps that should be executed in order to test a particular functionality. Moreover, tester needs to figure out loose points in the system from where he/she can discover defects. All these activities are highly imaginative and a tester is supposed to possess above average (if not excellent) analytical skills.

We shall explain the testing activities parallel to development activities with the help of the following diagram

© Copyright Virtual University of Pakistan



Description

- Functional specification document is the starting point, base document for both testing and the development
- Right side boxes describe the development, whereas, left side boxes explain the testing process
- Development team is involved into the analysis, design and coding activities.
- Whereas, testing team too is busy in analysis of requirements, for test planning, test cases and test data generation.
- System comes into testing after development is completed.
- Test cases are executed with test data and actual results (application behavior) are compared with the expected results,
- Upon discovering defects, tester generates the bug report and sends it to the development team for fixing.
- Development team runs the scenario as described in the bug report and try to reproduce the defect.
- If the defect is reproduced in the development environment, the development team identifies the root cause, fixes it and sends the patch to the testing team along with a bug resolution report.
- Testing team incorporates the fix (checking in), runs the same test case/scenario again and verifies the fix.
- If problem does not appear again testing team closes down the defect, otherwise, it is reported again.

11.5 The Developer and Tester

Development	Testing
Development is a creative activity	Testing is a destructive activity
Objective of development is to show that the program works	Objective of testing is to show that the program does not work

© Copyright Virtual University of Pakistan

Scenarios missed or misunderstood during development analysis would never be tested correctly because the corresponding test cases would either be missing or would be incorrect.

The left side column is related to the development, and the right side describes the testing. Development is a creative process as developers have to build the system, whereas, testing is a destructive activity as the goal of a tester is to break the system to discover the defects. Objective of development is to show that the program works, objective of testing is to show that program does not work. However, FS is the base document for both of these activities.

Tester analyzes FS with respect to testing the system whereas; developer analyzes FS with respect to designing and coding the system. If developer does not understand the FS correctly then he cannot implement and test it right. Thus if the same person who has developed a system, tests it, chances of carrying the same misunderstanding in testing will be very high. Therefore, an independent testing can only prove his understanding wrong. Therefore, it is highly recommended that developer should not try to test his/her own work.

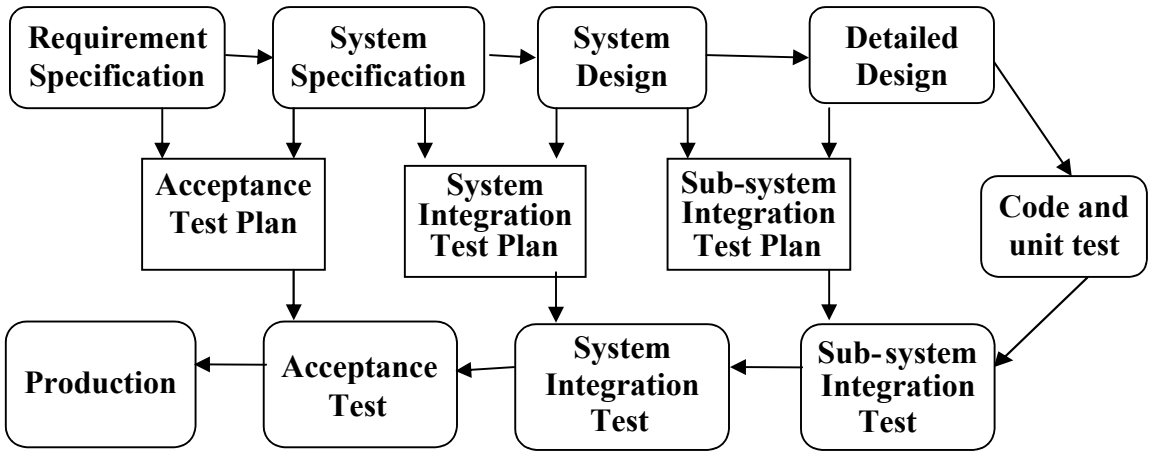
11.6 Usefulness of testing

Objective of testing is to discover and fix as many errors as possible before the software is put to use. That is before it is shipped to the client and the client runs it for acceptance. In software development organizations, a rift exists between the development and the testing teams. Often developers are found questioning about the significance or even need to have the testing resources in the project teams. Whoever doubts on the usefulness of the testing team should understand what could happen if the application is delivered to client without testing? At the best, the client may ask to fix all the defects (free of cost) he would discover during the acceptance testing. At the worst, probably he would sue the development firm for damages. However, in practice, clients are often seen complaining about the deliverables and a couple of defected deliverables are sufficient for breaking the relations next to the cancellation of contract.

Therefore, it should be well preserved among the community of developers that testers are essential rather inevitable. A good tester has a knack of smelling errors – just like auditors and it is for the good of the organization not to harm it.

11.7 Testing and software phases

With the help of the following diagram we shall explain different phases of testing



© Copyright Virtual University of Pakistan

Software development process diagram

Description of testing phases

- Unit testing – testing individual components independent of other components.
- Module testing – testing a collection of dependent components – a module encapsulates related components so it can be tested independently.
- Subsystem testing – testing of collection of modules to discover interfacing problems among interacting modules.
- System testing – after integrating subsystems into a system – testing this system as a whole.
- Acceptance test – validation against user expectations. Usually it is done at the client premises.
- Alpha testing – acceptance testing for customized projects, in-house testing for products.
- Beta testing – field testing of product with potential customers who agree to use it and report problem before system is released for general use

In the following two types of testing activities are discussed.

11.8 Black box testing

In this type of testing, a component or system is treated as a black box and it is tested for the required behavior. This type of testing is not concerned with how the inputs are transformed into outputs. As the system's internal implementation details are not visible to the tester. He gives inputs using an interface that the system provides and tests the output. If the outputs match with the expected results, system is fine otherwise a defect is found.

11.9 Structural testing (white box)

As opposed to black box testing, in structural or white box testing we look inside the system and evaluate what it consists of and how is it implemented. The inner of a system consists of design, structure of code and its documentation etc. Therefore, in white box testing we analyze these internal structures of the program and devise test cases that can test these structures.

Effective testing

The objective of testing is to discover the maximum number of defects with a minimum number of resources before the system is delivered to the next stage. Now the question arises here how to increase the probability of finding a defect?

As, good testing involves much more than just running the program a few times to see whether it works or not. A good tester carries out a thorough analysis of the program to devise test cases that can be used to test the system systematically and effectively. Problem here is how to develop a representative set of test cases that could test a complete program. That is, selection of a few test cases from a huge set of possibilities. What should be the sets of inputs that should be used to test the system effectively and efficiently?

String Equal Example

- For how many equal strings do I have to test to be in the comfortable zone?
- For how many unequal strings do I have to test to be in the comfortable zone?

© Copyright Virtual University of Pakistan

- When should I say that further testing is unlikely to discover another error? Testing types

To answer these questions, we divide a problem domain in different classes. These are called Equivalence Classes.

Lecture No. 38

Equivalence Classes or Equivalence Partitioning

Two tests are considered to be equivalent if it is believed that:

- if one discovers a defect, the other probably will too, and
- if one does not discover a defect, the other probably won't either.

Equivalence classes help you in designing test cases to test the system effectively and efficiently. One should have reasons to believe that the test cases are equivalent. As for this purpose, one would need to understand the system and see in how many partitions it can be divided. These partitions should be devised such that a clear distinction should be marked. Test cases written for one partition should not yield the same results when run for the second partition as otherwise these two partitions should become one. However, finding equivalence classes is a subjective process, as two people analyzing a program will probably come up with different sets of equivalence classes .

Equivalence partitioning guidelines

- Organize your equivalence classes. Write them in some order, use some template, sequence, or group them based on their similarities or distinctions. These partitions can be hierarchical or organized in any other manner.
- Boundary conditions: determine boundary conditions. For example, adding in an empty linked list, adding after the last element, adding before the first element, etc.
- You should not forget invalid inputs that a user can give to a system. For example, widgets on a GUI, numeric instead of alphabets, etc.

Equivalence partitioning example

In the following example, we shall see how equivalence partitions can be developed for a string matching function.

String matching

Organization

For equivalence partitions, we divide the problem in two obvious categories of equal strings and the other one for unequal strings. Within these equivalent partitions, further partitioning is done. Following is the description of the equivalence partitions and their test cases

Test cases for equivalence partitions

Equal

- Two equal strings of arbitrary length
  - All lower case "cat" "cat"
  - All upper case "CAT" "CAT"

© Copyright Virtual University of Pakistan

- Mixed case "Cat" "Cat"
- Numeric values "123" "123"
- Two strings with blanks only " " " "
- Numeric and character mixed "Cat1" "Cat1"
- Strings with special characters "Cat#1" "Cat#1"

- Two NULL strings "" ""

Unequal Strings

- Two different equal strings of arbitrary length
  - Two strings with different length "cat" "mouse"
  - Two strings of same length "cat" "dog"
- Check for case sensitivity
  - Same strings with different characters capitalized "Cat" "caT"
- One string is empty
  - First is NULL "" "cat"
  - Second is NULL "cat" ""

11.10 Basis Code Structures

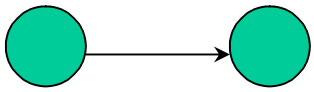
For structural testing it is important to know about basic coding structures. There are four basic coding structures sequence, if statement, case statement, and while loop. These four basic structures can be used to express any type of code.

Flow graph notation

In analysis and design, you have already seen the flow graph notation. This is used to describe flow of data or control in an application. However, we do not use flow graphs to describe decisions. That is, how a branch is taken is not shown in flow graphs.
In the following, we are using flow graph notation to describe different coding structures.

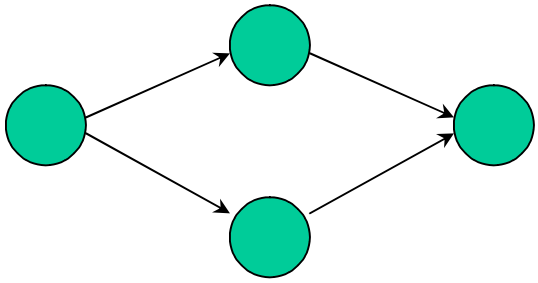
Sequence

Sequence depicts programming instructions that do not have branching or any control information. So we lump together several sequential instructions in one node of the graph.



If

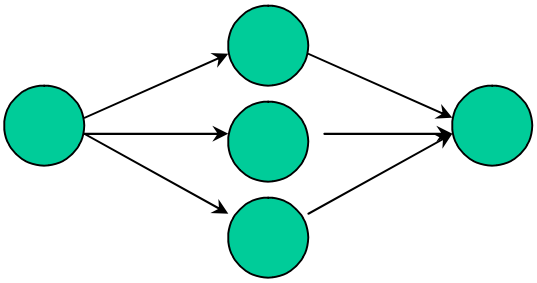
Second structural form is the If statement. In the following graph, the first node at the left depicts the if statement and the two nodes next to the first node correspond to the successful case (if condition is true) and unsuccessful case (if condition is false) consecutively. The control comes to the same instruction from either of these intermediate instructions.



© Copyright Virtual University of Pakistan

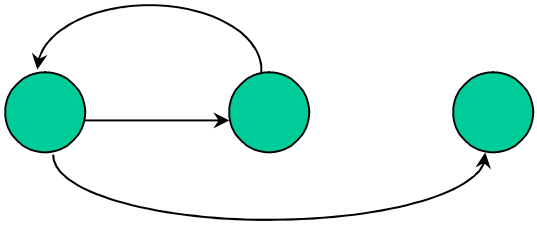
Case

In Case statement, control can take either of several branches (as opposed to only two in If statement.) First node represents the switch statement (C/C++) and nodes in middle correspond to all different cases. Program can take one branch and result into the same instruction.



While

A while loop structure consists of a loop guard instruction through which the iteration in the loop is controlled. The control keeps iterating in the loop as long as the loop guard condition is true. It branches to the last instruction when it becomes false.



Flow graph for bubble sort

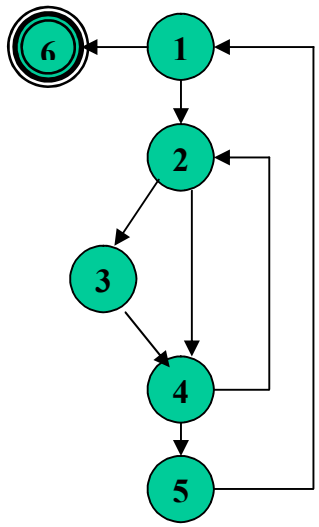
In the following example, code is given for a bubble sort function. The diagram opposite to the code is the corresponding flow graph model. It consists of six nodes. Node one is the while loop instruction that contains loop guard and node six is the ending instruction of the while loop. Node two corresponds to the for loop instructions. Node three corresponds to the swapping instruction. Nodes five and six correspond to the last instructions of the if statement and the for loop consecutively.

Point to note here is the assignment of node numbers to program instructions. As, the corresponding flow graph model would consist of nodes that correspond to instructions which are major decision points in the code. For example, when this function will be invoked, control will certainly come to the while loop instruction and at the minimum it will traverse from node one to node six. However, if control enters into the while loop even for a single iteration, it will traverse through nodes two, four and five for certain and node three too if it enters for loop and check in the if condition is true.

So all these combinations are to be tested during white box testing.

© Copyright Virtual University of Pakistan

```
sorted = false;
while (!sorted) {                  //1
    sorted = true;
    for (i=0; i < N-1; i++) {      //2
        if a[i] > a[i+1] {
            swap(a[i], a[i+1]);    //3
            sorted = false;
        }                          //4
    }                              //5
}                                  //6
```



Paths

Following are possible paths from starting to the end of this code.
Path1: 1-6
Path2: 1-2-3-4-5-1-6
Path3: 1-2-4-5-1-6
Path4: 1-2-4-2-3-4-5-6-1

Lecture No. 39

White box testing

As described in the section above, in white box testing we test the structure of the program. In this technique the test cases are written in a manner to cover different possibilities in code. Below are described three coverage schemes.

Coverage

- **Statement Coverage**: In this scheme, statements of the code are tested for a successful test that checks all the statements lying on the path of a successful scenario.
- **Branch Coverage**: In this scheme, all the possible branches of decision structures are tested. Therefore, sequences of statements following a decision are tested.
- **Path Coverage**: In path coverage, all possible paths of a program from input instruction to the output instruction are tested. An exhaustive list of test cases is generated and tested against the code.

White Box Testing Example

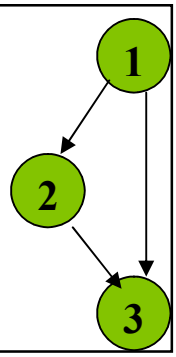
With the help of the following example, we shall see how many test cases are generated for each type of coverage schemes.

© Copyright Virtual University of Pakistan

```
if (a = = b)          //1

    c = d;            //2

a++;                  //3
```



Statement coverage:

- a=1,b=1
- If a==b then statement 2, statement 3

Branch coverage

- Statement 1: two braches 1-2, 1-3
- Test case 1: if a =1, b=1 then statement 2
- Test case 2: if a=1,b=2: then statement 3

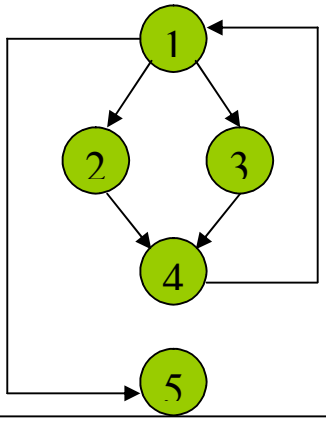
Path coverage

Same as branch testing

Paths in a program containing loops

Now we shall apply the path coverage scheme on a piece of code that contains a loop statement and see how many test cases can possibly be developed.

```
for (i = 0; i < N; i++) {          //1
  if (condition1)
        // do something here       //2
  else
        // do something here       //3
  // something here                //4
}
                                   //5
```



Paths

The following is an analysis of the above-mentioned code and the flow diagram. It determines the number of paths against different iterations of the loop.

- N = 0: If the control does not enter into the loop then only one path will be traversed. It is 1-5.
- N=1: Two different paths can possibly be traversed (depending on condition).
  - 1-2-4-1-5
  - 1-3-4-1-5
- N=2: Four possible paths can be traversed.
  - 1-2-4-1-2-4-1-5
  - 1-2-4-1-3-4-1-5

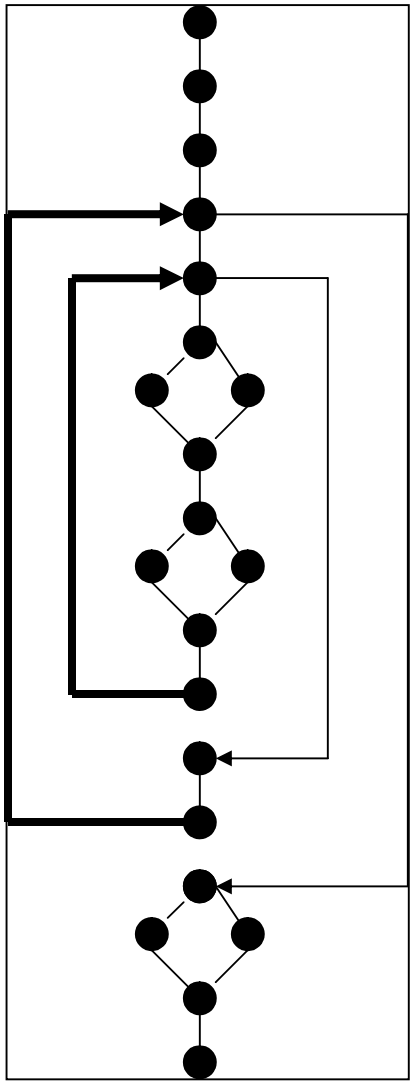
© Copyright Virtual University of Pakistan

- 1-3-4-1-2-4-1-5
- 1-3-4-1-3-4-1-5

- Generalizing the relation between loop variable N and the number of possible paths, for the value of N, 2^N paths are possible
  - Thus if N = 20 it means more then 1 million paths are possible.

Flow graph of a hypothetical program

Following is a flow graph of a hypothetical program whose structure is such that it consists of two loops, one embedded in the other. Left side of the diagram signifies the loops where inner loop seem to be containing two if statements. Second loop has a branch and then program finishes.
Simple graph contains 1852 paths with each loop not iterated more than twice



Thus, the number of paths in a program that contains loops tends to infinity. It is impossible to conduct exhaustive testing of a program that may consist of infinite number of test cases. The question arises, how many test cases need to be executed in order to test all the major scenarios in the code at least once? The answer is, calculate the cyclomatic

© Copyright Virtual University of Pakistan

complexity of the code. This will give us a number that corresponds to the total number of test cases that need to be generated in order to test all the statements and branches in the code at least once.

Cyclomatic complexity

The concept of cyclomatic complexity is extremely useful in white box testing when analyzing the relative complexity of the program to be tested. It revolves around independent paths in a program which is any path through the program (from start to end) that introduces at least one new set of processing statements or a new condition. An independent path covers statements and branches of the code.

Cyclomatic complexity is a quantitative measure of the logical complexity of a program. It defines number of independent paths in the basis set of a program. It provides an upper bound for the number of tests that must be conducted to ensure that all statements and branches have been executed at least once.

Cyclomatic Complexity, V(G), for a flow graph G is defined as:

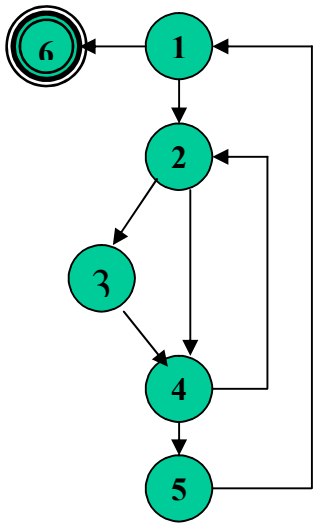
$$V(G) = E - N + 2$$

Where E is the number of edges and N is the number of nodes in the flow graph G. Cyclomatic complexity provides us with an upper bound for the number of independent paths that comprise the basis set.

Cyclomatic Complexity of a Sort Procedure

Following is the same bubble sort program that we discussed above. This time we shall calculate its cyclomatic complexity and see how many test cases are needed to test this function.

```
Sorted = FALSE;
while (!sorted) {                  //1
   sorted = TRUE;
   for (i=0; i < N-1; i++) {       //2
      if a[i] > a[i+1] {
         swap(a[i], a[i+1]);       //3
         sorted = FALSE;
      }                            //4
   }                               //5
}                                  //6
```



Cyclomatic complexity

- Number of edges = 8
- Number of nodes = 6
- C(G) = 8-6+2 = 4

Paths to be tested

- Path1: 1-6

© Copyright Virtual University of Pakistan

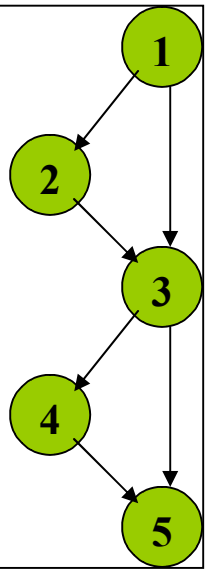
- Path2: 1-2-3-4-5-1-6
- Path3: 1-2-4-5-1-6
- Path4: 1-2-4-2-3-4-5-6-1

Infeasible paths

Infeasible path is a path through a program which is never traversed for any input data.

Example

```
if (a == b)          //1
    c = c-1;         //2
if (a != b)          //3
    c = c+1;         //4
                     //5
```



In the above-mentioned example, there are two infeasible paths that will never be traversed.

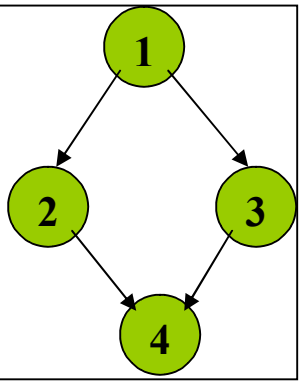
- Path1: 1-2-3-4-5
- Path2: 1-3-5

A good programming practice is such that minimize infeasible paths to zero. It will reduce the number of test cases that need to be generated in order to test the application. How can we minimize infeasible paths, by simply using else part with the if statement and avoid program statements as given above.

Modified code segment

Infeasible paths can be analyzed and fixed.

```
if (a == b)          //1
      c = c-1;       //2
else
      c = c+1;       //3
                     //4
```



There are no infeasible paths now!

© Copyright Virtual University of Pakistan

Lecture No. 40

11.11 Unit testing

A software program is made up of units that include procedures, functions, classes etc. The unit testing process involves the developer in testing of these units. Unit testing is roughly equivalent to chip-level testing for hardware in which each chip is tested thoroughly after manufacturing. Similarly, unit testing is done to each module, in isolation, to verify its behaviour. Typically the unit test will establish some sort of artificial environment and then invoke routines in the module being tested. It then checks the results returned against either some known value or against the results from previous runs of the same test (regression testing). When the modules are assembled we can use the same tests to test the system as a whole.

Software should be tested more like hardware, with

- Built-in self testing: such that each unit can be tested independently
- Internal diagnostics: diagnostics for program units should be defined.
- Test harness

The emphasis is on built in testability of the program units from the very beginning where each piece should be tested thoroughly before trying to wire them together.

Unit Testing Principles

- In unit testing, developers test their own code units (modules, classes, etc.) during implementation.
- Normal and boundary inputs against expected results are tested.
- Thus unit testing is a great way to test an API.

Quantitative Benefits

- **Repeatable**: Unit test cases can be repeated to verify that no unintended side effects have occurred due to some modification in the code.
- **Bounded**: Narrow focus simplifies finding and fixing defects.
- **Cheaper**: Find and fix defects early

Qualitative Benefits

- **Assessment-oriented**: Writing the unit test forces us to deal with design issues - cohesion, coupling.
- **Confidence-building**: We know what works at an early stage. Also easier to change when it's easy to retest.

Testing against the contract (Example)

When we write unit tests we want to write test cases that ensure a given unit honors its contract. This will tell us whether the code meets the contract and whether the contract means what we think the unit is supposed to do in the program.

Contract for square root routine

```
result = squareRoot(argument);
assert (abs (result * result – argument) < epsilon);
```


© Copyright Virtual University of Pakistan

The above contract tells us what to test:

- Pass in a negative argument and ensure that it is rejected
- Pass in an argument of zero to ensure that it is accepted (this is a boundary value)
- Pass in values between zero and the maximum expressible argument and verify that the difference between the square of the result and the original argument is less than some value epsilon.

When you design a module or even a single routine, you should design both its contract and the code to test that contract. By designing code to pass a test that fulfill its contract, you might consider boundary conditions and other issues that you wouldn't consider otherwise. The best way to fix errors is to avoid them in the first place. By building the tests *before* you implement the code you get to try out the interface before you commit to it.

Unit Testing Tips

Unit test should be conveniently located

- For small projects you can imbed the unit test for a module in the module itself
- For larger projects you should keep the tests in the package directory or a /test subdirectory of the package

By making the code accessible to developers you provide them with:

- Examples of how to use all the functionality of your module
- A means to build regression tests to validate any future changes to the code

You can use the main routine with conditional compilation to run your unit tests.

11.12 Defect removal efficiency

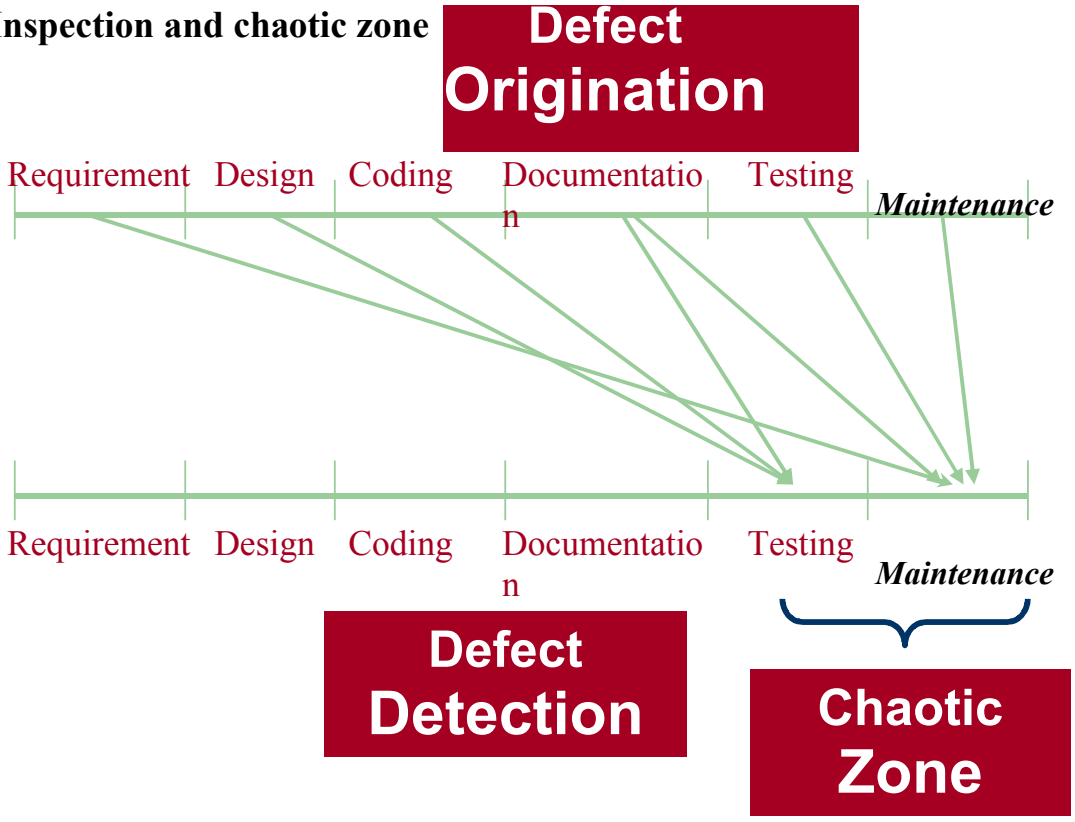
Is the ability to remove defects from the application. We shall further elaborate this idea with the help of the above-mentioned table.

Design Inspection					●		●	●	●
Code Inspection				●		●		●	●
Quality Assurance			●			●	●		●
Testing		●	●	●	●	●	●	●	●
Worst	**30%**	**37%**	**50%**	**55%**	**65%**	**75%**	**77%**	**85%**	**95%**
Median	**40%**	**53%**	**65%**	**70%**	**80%**	**87%**	**90%**	**97%**	**99%**
Best	**50%**	**60%**	**75%**	**80%**	**87%**	**93%**	**95%**	**99%**	**99.9%**

The above table depicts data that was published after analyzing 1500 projects. In these projects, four different types of quality assurance mechanisms were employed. It is evident from this table that testing alone can only remove 53% of defects. However, testing and quality assurance mechanisms combine yield up to 65% efficiency. Whereas, if we combine code inspection and testing together, results are up to 75%. Similarly, design inspections and testing yield up to 80%. Moreover, combining design inspections, quality assurance and testing results are up to 95%. If all four techniques are combined, results are up to 99.9%.

© Copyright Virtual University of Pakistan

Inspection and chaotic zone



In this diagram, a chaotic zone has been defined. In fact, if defects are not discovered and fixed at the appropriate stage, then at the testing and maintenance phases, these defects are piled up. Therefore, a chaotic zone is formed for the testing and the development teams as the number of defects which are piled up, destabilize the application and it becomes extremely hard to fix all these defects as some of these bugs may involve changes in requirements and design. At testing or the maintenance phases, fixing a defect in requirements or design becomes extremely expensive, as underlying code will have to be changed as well.

If we combine the results of the above two diagrams, it is evident that testing alone does not suffice. We need to employ inspection techniques and combine them with testing to increase the effectiveness of defect removal efficiency.

11.13 Defect origination

In inspections the emphasis is on early detection and fixing of defects from the program. Following are the points in a development life cycle where defects enter into the program.

- Requirements
- Design
- Coding
- User documentation
- Testing itself can cause defects due to bad fixes
- Change requests at the maintenance or initial usage time

© Copyright Virtual University of Pakistan

It is important to identify defects and fix them as near to their point of origination as possible.

Lecture No. 41

11.14 Inspection versus Testing

Inspections and testing are complementary and not opposing verification techniques. Both should be used during the verification and validation process. Inspections can check conformance with a specification but not conformance with the customer's real requirements. Inspections cannot check non-functional characteristics such as performance, usability, etc. Inspection does not require execution of program and they maybe used before implementation. Many different defects may be discovered in a single inspection. In testing, one defect may mask another so several executions are required. For inspections, checklists are prepared that contain information regarding defects. Reuse domain and programming knowledge of the viewers likely to help in preparing these checklists. Inspections involve people examining the source representation with the aim of discovering anomalies and defects. Inspections may be applied to any representation of the system (requirements, design, test data, etc.) Thus inspections are a very effective technique for discovering errors in a software program.

Inspection pre-conditions

A precise specification must be available before inspections. Team members must be familiar with the organization standards. In addition to it, syntactically correct code must be available to the inspectors. Inspectors should prepare a checklist that can help them during the inspection process.

Inspection checklists.

Checklist of common errors in a program should be developed and used to drive the inspection process. These error checklists are programming language dependent such that the inspector has to analyze major constructs of the programming language and develop checklists to verify code that is written using these checklists. For example, in a language of weak type checking, one can expect a number of peculiarities in code that should be verified. So the corresponding checklist can be larger. Other example of programming language dependant defects are defects in variable initialization, constant naming, loop termination, array bounds, etc.

Inspection Checklist

Following is an example of an inspection checklist.

Exception management faults	• Have all possible error conditions been taken into account?
Fault Class	• Inspection Check

© Copyright Virtual University of Pakistan

Data faults	• Are all program variables initialized before their values are used? • Have all constants been named? • Should the lower bound of arrays be 0, 1, or something else? • Should the upper bound of arrays be size or size -1? • If character strings are used, is a delimiter explicitly assigned?
Control faults	• For each conditional statement, is the condition correct? • Is each loop certain to terminate? • Are compound statements correctly bracketed? • In case statements, are all possible cases accounted for?
Input/Output faults	• Are all input variables used? • Are all output variables assigned a value before they are output?
Interface faults	• Do all function and procedure calls have correct number of parameters? • Do formal and actual parameters types match? • Are the parameters in right order? • If components access shared memory, do they have the same model of shared memory structure?
Storage management faults	• If a linked structure is modified, have all links been correctly assigned? • If dynamic storage is used, has space been allocated correctly? • Is space explicitly de-allocated after it is no longer required?

In the checklist mentioned above, a number of fault classes have been specified and their corresponding inspection checks are described in the column at the right side. This type of checklist helps an inspector to look for specific defects in the program. These inspection checks are the outcomes of experience that the inspector has gained out of developing or testing similar programs.

11.15 Static analyzers

Static analyzers are software tools for source text processing. They parse the program text and try to discover potentially erroneous conditions and bring these to the attention of the verification and validation team. These tools are very effective as an aid to inspections. But these are supplement to but not a replacement for inspections.

© Copyright Virtual University of Pakistan

Checklist for static analysis

Data faults	• variable used before initialization • variable declared but never used • variables assigned twice but never used between assignments • possible array bound violations • undeclared variables
Control faults	• unreachable code • unconditional branches into loops
Input/Output faults	• variable output twice with no intervening assignment
Storage Management fault	• unassigned pointers • pointer arithmetic

© Copyright Virtual University of Pakistan

Lecture No. 42

Debugging

12.1 Debugging

As Benjamin Franklin said, “in this world, nothing is certain but death and taxes.” If you are in the software development business, however, you can amend that statement. Nothing in life is certain except death, taxes, and software bugs. If you cryogenically freeze yourself, you can delay death indefinitely. If you move to a country with no income tax, you can avoid paying taxes by not buying anything. If you develop software, however, no remedy known to mankind can save you from the horror of software bugs.

What is a Bug?

We call them by many names: software defects, software bugs, software problems, and even software “features.” Whatever you want to call them, they are things the software does that it is not supposed to do (or, alternatively, something the software doesn’t do that it is supposed to). Software bugs range from program crashes to returning incorrect information to having garbled displays.

12.2 A Brief History of Debugging

It would appear that as long as there have been computers, there have been computer bugs. However, this is not exactly true. Even though the earliest known computer programs contained errors, they were not, at that time, referred to as “bugs.” It took a lady named Admiral Grace Hopper to actually coin the term “bug.”

After graduating from Vassar in 1928, she went to Yale to receive her master’s degree in mathematics. After graduating from Yale, she worked at the university as a mathematics professor. Leaving Yale in 1943, with the onset of World War II, Mrs. Hopper decided to work for the Navy. Mrs. Hopper’s first assignment was under Commander Howard Aiken at Howard University, working at the Bureau of Ordinance Computation. She was a programmer on the Mark II, the world’s first automatically sequenced digital computer. The Mark II was used to determine shooting angles for the big guns in varying weather conditions during wartime.

It was during her term with the Mark II that Hopper was credited with coining the term “bug” for a computer problem. The first “bug” was actually a moth, which flew through an open window and into one of the Mark II’s relays. At that time, physical relays were used in computers, unlike digital components we use today. The moth shorted out across two contacts, temporarily shutting down the system. The moth would later be removed (de-bugged?) and pasted into the logbook of the project. From that point on, if her team was not producing numbers or working on the code, they claimed to be “debugging” the system.

From that auspicious beginning, computer debugging developed into something of a hit-or-miss procedure for quite a few years. Early debugging efforts mostly centered around either data dumps of the system or used output devices, such as printers and display lights, to indicate when an error occurred. Programmers would then step through the code line by line until they could determine the location of the problem. The next step in the

© Copyright Virtual University of Pakistan

evolution of debugging came with the advent of command-line debuggers. These simple programs were an amazing step forward for the programmers. Although difficult to use, even at the time, these programs represented the first real attempt to turn debugging from a hit-or-miss proposition into a reproducible process.
Once the debugger became a part of the programmer's arsenal, the software world began to consider other ways that programs could be more easily debugged. Software projects starting getting bigger, and the same techniques that worked well for small projects no longer worked when the program reached a certain size.

Importance of Debugging

As we mentioned earlier in this course, one of the prime objectives of software engineering is to develop cost effective software applications. According to a survey, when a software application is in the maintenance phase, 20% of its lifecycle cost is attributed towards the defects which are found in the software application after installation. Please bear in mind that the maintenance is the phase in which 2/3rd of the overall software cost incurs. Therefore, 20% of the 2/3rd cost is again a huge cost and we need to understand why this much cost and effort is incurred. In fact, when a software application is installed and being used, any peculiarity in it can cost a lot of direct and indirect damages to the organization. A system downtime is the period in which tremendous pressure is on developers end to fix the problem and make the system running again. In these moments, every second costs hugs losses to the organization and it becomes vital to find out the bug in the software application and fix it. Debugging techniques are the only mechanism to reach at the code that is malfunctioning. In the following subsection, we shall discuss an incident that took place in 1990 and see how much loss the company had to suffer due to a mere bug in the software application.

12.4 Problem at AT&T

In the telecommunications industry, loss of service is known as an outage. For most of us, when as outage occurs, we lose our telephone service; we cannot make calls and we cannot receive calls. Outages are accepted and expected hazards in the industry.
On January 15. 1990, AT&T had a US wide telephone system outage that lasted for nine hours. The cause was due to a program error in the software that was meant to make the system more efficient. Eight years later, on April 13, 1998, AT&T suffered another massive failure in its frame-relay network, which affected ATM machines, credit card transactions and other business data services. This failure lasted 26 hours. Again, the bug was introduced during a software upgrade.

Description of the Problem

The code snippet that caused the outage is illustrated as follows

```
1. do {
2.     . . .
3.     switch (expression) {
4.         case 0: {
5.             if (some_condition) {
```


© Copyright Virtual University of Pakistan

```
6.                  . . .
7.                  break;
8.              } else {
9.                  . . .
10.             }
11.           . . .
12.           break;
13.       }
14.     . . .
15.   }
16.   . . .
17. } while (some_other_condition);
```

In this case the break statement at line 7 was the culprit. As implemented, if the logical_test on line 5 was successful, the program should have proceeded to line 6 to execute those statements. When the program stepped to line 7, the break statement caused the program to exit the "switch" block between line 3 and line 15, and proceeded to execute the codes in line 16. However, this path of execution was not the intention of the programmer. The programmer intended the break statement in line 7 to break the if-then clause; so, after the program executed line 7, it was supposed to continue execution in line 11.

AT&T statement about the Problem

"We believe that the software design, development, and testing processes we use are based on solid, quality foundations. All future releases of software will continue to be religiously tested. We will use the experience we've gained through this problem to further improve our procedures."

We do not believe we can fault AT&T's software development process for the 1990 outage, and we have no reason to believe that AT&T did not rigorously test its software update. In hindsight, it is easy to say that if the developers had only tested the software, they would have seen the bug. Or that if they had performed a code inspection, they would have found the defect. Code inspection might have helped in this case. However, the only way the code inspection could have uncovered this bug is if another engineer saw this particular line of code and asked the original programmer if that was his or her intension. The only reason that the code reviewers might have asked this question is if they were familiar with the specifications for this particular code block.

12.5Art and Science of Debugging

Debugging is taken as an art but in fact it is a scientific process. As people learn about different defect types and come across situations in which they have to debug the code, they develop certain heuristics. Next time they come across a similar situation, they apply those heuristics and solve the problem in lesser time and with a lesser effort. While discussing the debugging process we discuss the phenomenon of "you miss the obvious". When a person writes a code, he develops certain impression about that code. One can term this impression as a personal bias that the developer builds towards his creation the

© Copyright Virtual University of Pakistan

"code" and when he has to check this code, he can potentially miss out obvious mistakes due to this impression or bias. Therefore, it is strongly recommended that in order to reach to a defect in the code, one needs "another pair of eyes". That is, start discovering the defect by applying your own heuristics and if you could reach to the problem, fine, otherwise ask a companion to help you in this process. We shall further elaborate this idea based on the following example.

Program at Bulletin Board Example

Following piece of code was once placed on a bulletin board making an invitation to people to discover the problem in this code. Please look at this code and discover the problem

```
1. while (i =0; i < 10; i++) {
2.     cout << i << newl;
3. }
```

Well if you could not guess it by now, the problem lies with the syntax of the while loop. This is so obvious that almost everyone forgot to ponder upon when placed on the bulletin board of a university. The loop is "while" but the syntax used is a "for" loop.

In order to reach such defects, one needs a scientific approach to check and verify the code methodically. Based on this discussion, we are now in a position to introduce a few classes of bugs to the reader. This is not an exhaustive list since there could be a number of other classes of bugs as well but the following classes will help the reader know about some well known bugs that we usually find in the code.

Lecture No. 43

12.6 Bug Classes

Memory and resource leak

A memory leak bug is one in which memory is somehow allocated from either the operating system or an internal memory "pool", but never deallocated when the memory is finished being used.

Symptoms

- System slowdowns
- Crashes that occur "randomly" over a long period of time

Example 1

Let's take a look at a simple memory leak error that can occur trivially in a C program. This type of code is found in hundreds of programs across the programming spectrum. It illustrates the simplest possible case of a memory leak, which is when memory is allocated and not deallocated in all cases. This particular form of the bug is the most frustrating because the memory is usually deallocated, but not always

© Copyright Virtual University of Pakistan

```
char *buffer = new char[kMaxBufferSize+1];
memset(buffer,0,kMaxBufferSize+1);

// Do some stuff to fill and work with the character buffer

if (IsError(nCondition)) // Did we get an error in the processing piece?
{
  Message(("An error occured.
            Skipping final stage"));
  return FailureCode;
}

// Final stage of code

// Free up all allocated memory

delete buffer;
return okCode;
```

Note that in many cases, this code works perfectly. If no error occurs in the processing stage (which is the norm) the memory is freed up properly. If, however, the processing stage encounters an error, the memory is not freed up and a leak occurs.

Example 2

The following code snippet from some old C++ code contains a slightly more insidious bug. It illustrates the point well.

```
Class Foo
{
  private:
    int nStringLength;
    char *sString;
  public:
    Foo()
    {
      nStringLength = 20; //Default
      sString = new
      char[nStringLength + 1];
    }
    ~Foo()
    {
      delete sString;
    }
    void SetString(const char
                    *inString)
    {
```


© Copyright Virtual University of Pakistan

```
        sString = new char
                  [strlen(inString+1)];
        if(inString == NULL)
          return;
        strncpy(sString, inString,
                  strlen(inString));
        nStringLength =
                strlen(inString)+1;
    }
};
```

Let's discuss what happens. In most cases, the Foo object is created and nothing bad happens. If, however, you call the SetString method, all bets are off. The previously allocated string is overwritten by the new allocation, and the old allocated memory goes nowhere. We have an instant memory leak. Worse, we can do this multiple times if we accidentally call the method with a NULL string because it first allocates the block, and then checks to see if the input string was correct. Don't do things like this. If you see an allocation in a class, check to see if the string can be allocated before it gets to that point.

Logical Errors

A logical error occurs when the code is syntactically correct but does not do what you expect it to do.

Symptoms

- The code is misbehaving in a way that isn't easily explained.
- The program doesn't crash, but the flow of the program takes odd branches through the code.
- Results are the opposite of what is expected.
- Output looks strange, but has no obvious symptoms of corruption.

Example

```
// Make sure that the input is valid. For this value, valid ranges are 1-10 and 15-20

if((input >= 1 && input <= 10) &&
  (input >= 15 && input <= 20))
  {
    // Do something for valid case
  }
else
  {
    // Do something for invalid case
  }
```

In order for the code to enter the valid case, the number must be between 1 and 10 and be between 15 and 20. If a number is between 1 and 10, how can it possibly be between 15 and 20 as well? Seems unlikely, doesn't it? This is a typical logical error.

Coding errors

© Copyright Virtual University of Pakistan

A coding error is a simple problem in writing the code. IT can be a failure to check error returns, a failure to check for certain valid conditions, or a failure to take into account other parts of the system. Yet another form of a coding error is incorrect parameter passing and invalid return type coercion.

Symptoms

- Unexpected errors in black box testing.
- The errors that unexpectedly occur are usually caused by coding errors.
- Compiler warnings.
- Coding errors are usually caused by lack of attention to details.

Example

In the following example, a function accepts an input integer and converts it into a string that contains that integer in its word representation.

```
void convertToString(int InInteger,
      char* OutString, int* OutLength)
{
  switch(InInteger){
     case 1: OutString = "One";OutLength = 3;
               break;
     case 2: OutString = "Two";OutLength = 3;
               break;
     case 3: OutString = "Three";OutLength = 5;
               break;
     case 4: OutString = "Four";OutLength = 4;
               break;
     case 5: OutString = "Five";OutLength = 4;
               break;
     case 6: OutString = "Six";OutLength = 3;
               break;
     case 7: OutString = "Seven";OutLength = 5;
               break;
     case 8: OutString = "Eight";OutLength = 5;
               break;
     case 9: OutString = "Nine";OutLength = 4;
               break;

     }
}
```

There are a few things to notice in the preceding code. The ConvertToString function does not handle all cases of inputs, which becomes very obvious you pass a zero value. Worse, the function does not initialize the output variables, leaving them at whatever they happened to be when they came into the function. This isn't a problem for the output string, necessarily, but it will become a serious issue for the output length.

Memory over-runs

© Copyright Virtual University of Pakistan

a memory overrun occurs when you use memory that does not belong to you. This can be caused by overstepping an array boundary or by copying a string that is too big for the block of memory it is defined to hold. Memory overruns were once extremely common in the programming world because of the inability to tell what the actual size of something really was.

Symptoms

- Program crashes quite regularly after a given routine is called, that routine should be examined for a possible overrun condition.
- If the routine in question does not appear to have any such problem the most likely cause is that another routine, called in the prior sequence, has already trashed variables or memory blocks.
- Checking the trace log of the called routines leading up to one with the problem will often show up the error.

Example

This particular example shows not only a memory overrun, but also how most programmers "fix" problems in an application. The ZeroArray function steps all over the array boundaries by initializing 100 separate entries. The problem is that that particular array only has 50 slots available in its allocation. What happens at that point is that the function goes past the end of the array and starts to walk on things beyond its control.

```
const kMaxEntries = 50;
int gArray[kMaxEntries];
char szDummyBuffer[256];
int nState = 10;

int ZeroArray (int *pArray)
{
  for (inti=0;i<100;++i)
    pArray[i] = 0;
}
```

Loop Errors

- Loop errors break down into several different subtypes.
- They occur around a loop construct in a program.
- Infinite loops, off-by-one loops, and improperly exited loops.

Symptoms

- If your program simply locks up, repeatedly displays the same data over and over, or infinitely displays the same message box, you should immediately suspect an infinite loop error.
- Off-by-one loop errors are quite often seen in processes that perform calculations.
- If a hand calculation shows that the total or final sum is incorrect by the last data point, you can quickly surmise that an off-by-one loop error is to blame.
- Likewise, if you were using graphics software and saw all of the points on the screen, but the last two were unconnected, you would suspect an off-by-one error.

© Copyright Virtual University of Pakistan

- Watching for a process that terminates unexpectedly when it should have continued.

Example

```
bool doneFlag;
doneFlag = false;
while(!doneFlag){
  ...
  if( impossibleCondition )
    doneFlag = true;
}
```

The preceding code fragment contains an indirect looping error such that the loop will continue until the impossibleCondition becomes true which is impossible to happen.

```
int anArray[50];
int i;
i = 50;
while(i >= 0){
  anArray[i] = 0;
  i = i - 1;
}
```

In this example, the programmer was trying to be smart. He worked his way backward though the array, assuming that this way there would be no chance of an error. Of course, if you examine the loop, you will find that it performed 51 times. There are only fifty elements in the array. This will certainly lead to a problem later on in the program, if it doesn't first trigger an immediate error from the system.

```
int nIndex = 0;
for(int i=0; i<kMaxIterations; ++i)
{
  while (nIndex < 20)
  {
    ComputeSomething(i*20 + nIndex);
    nIndex ++;
  }
}
```

The final situation is an improper exit condition for a loop. This is most easily illustrated using a set of nested loops. In the above example, we are trying to compute something within the inner loop for some number of iterations. The code appears to do what we said it should do. It computes whatever it is we are computing in the inner loop for each iteration of the outer loop, 20 times. The problem, however, is that the exit condition simply says that nIndex should be less than 20. The first time through the outer loop, this variable will become 21, and the inner loop will exit. This is correct, and exactly the way

© Copyright Virtual University of Pakistan

you would expect this to happen. The problem, however, is that the next time through the loop, the inner loop will not be executed at all.

Pointer errors

- A pointer error is any case where something is being used as an indirect pointer to another item.
- Uninitialized pointers: These are pointers that are used to point at something, but we fail to ever assign them something to point at.
- Deleted pointers, which continue to be used.
- An Invalid pointer is something that is pointing to a valid block of memory, but that memory does not contain the data you expect it to.

Symptoms

- The program usually crashes or behaves in an unpredictable and baffling way.
- You will generally observe stack corruptions, failure to allocate memory, and odd changing of variable values.
- Changing a single line of code can change where the problem occurs.
- If the problem "goes away" when you place a print statement or new variable into the code that you suspect contains the problem.

Example

Let's take a look at one of the most common forms of the pointer error, using a pointer after it has been deleted. As you can see by the following example, it is not always clear when you are doing this incorrectly:

```
void cleanup_function(char *ptr)
{
  SaveToDisk(ptr);
  delete ptr;
}

int func()
{
  char *s = new char[80];
  cleanup_function(s);
  delete s;
}
```

In this example, the programmer meant to be neat and tidy. He probably wrote the code originally to allocate the pointer at the top of this function, and then by habit put in a deallocation at the bottom of the function. This is, after all, good programming practice to avoid memory leaks. The problem is introduced with th3e cleanup_function routine. This function was probably written at a later time by another developer and might have been used to ensure that all allocated pointers were stored to the permanent drive and then freed up to avoid any possible leaks. In fact, the code in question comes from a set of functions that was reused from a memory pool system that saved its data to disk before destroying the pointer, so that another pointer could be retrieved from the persistent pool.

© Copyright Virtual University of Pakistan

The problem is, the reuse did not take into account the existing system. This problem is common when reusing only parts of a system.
When the second delete occurs at the bottom of func(), the results are unpredictable. At best, the delete function recognizes that this pointer has been deleted already and doesn't do anything. At worst, memory is corrupted and you are left with a memory crash somewhat later in the program, which will be very difficult to find and fix.

Boolean bugs

Boolean bugs occur because the mathematical precision of Boolean algebra has virtually nothing to do with equivalent English words.
When we say "and", we really mean the boolean "or" and vice versa.

Symptoms

- When the program does exactly the opposite of what you expect it to. For example, you might have thought you needed to select only one entry from a list in order to proceed. Instead, the program will not continue until you select more than one. Worse, it keeps telling you to select only one value.
- For true/false problems, you will usually see some sort of debug output indicating an error in a function, only to see the calling function proceed as though the problem had not occurred.

Examples

The following code contains a simple example of a function that returns a counter-intuitive Boolean value and shows how that value leads to problems in the rest of the application code. This case performs an action and indicates to the user whether or not the action succeeded.

```
int DoSomeAction(int InputNum)
{
  if(InputNum < 1 || InputNum  > 10)
       DoSomeAction = 0;
  else
  {
    PerformTheAction(InputNum);
    DoSomeAction = NumberOfAction + 1;
    NumberOfAction = NumberOfAction + 1;
  }
  return DoSomeAction;
}
```

After looking at the function, ask yourself this – What is the return value in the case of success? What would it be in the case of failure? How would you know this by simply glancing at the function declaration?

```
int Value;
int Ret;

Value = 11;
```


© Copyright Virtual University of Pakistan

```
Ret = DoSomeAction(Value);
if(Ret)
  cout <<Error in DoSomeAction<<;
```

If you were debugging this application, you might start out by wondering why it didn't work. Further investigation would show that the error statement is never triggered in spite of the fact that the routine did, in fact, encounter an error.. Why did this happen? It happened because the routine didn't return nonzero value for failure, it returned nonzero for success. This Boolean value was not intuitive given the condition of the function and led to a poor assumption by the programmer who wrote the code that used it.

Lecture No. 44

12.7 Holistic approach

Holistic means

- Emphasizing the importance of the whole and the interdependence of its parts.
- Concerned with wholes rather than analysis or separation into parts

What this meant is that a holistic approach focuses on the entire system rather than whatever piece appears to be broken. Holistic medicine, for example, concerns itself with the state of the body as a whole, not the disease that is currently attacking it. Similarly, programmers and debuggers must understand that you cannot treat the symptoms of a problem, you must focus on the application system as a whole.

Now that you understand what holistic means, how do you apply holistic concepts to debugging the software programs and procedures? Let's start by thinking about what emphasizing the whole over the individual pieces means in debugging. When you are debugging an application, you often simply look at the problem reported by the user and how you can make that problem go away.

As a simple example, consider the case where the program mysteriously crashes at a particular point in the code each and every time the program is run. The point at which the code crashes makes no sense at all. For example, in C++, you might have a member function of a class that reads:

```
void SetX(int X)
{
  mX = X;
}
```

The program always crashes on the line that assigns X to the member variable mX(mX = X). Looking at this code snippet, we see very few valid ways in which the program could be encountering a problem at this point. Oddly, we might discover that inserting a message statement makes the problem stop occurring when the indicated steps are followed for reproducing the problem. Is the problem fixed?

© Copyright Virtual University of Pakistan

Most experienced programmers, managers, or debuggers would say that the problem is not fixed, although few could tell you exactly why that is the case. The fact is, in this case, we are directly treating the symptoms (the program crashing) rather than looking into the actual problem and ignoring the overall problem is endemic in our industry.

12.8 The debugging process

In normal circumstances, you will have a user description of the problem. This description might have been given to you directly by the user, or it might have been gathered by a customer support person or other no technical person, In any event, this data has to be considered suspiciously until you can get a first hand description of what really happened. First hand accounts of the problem are always useful, so be sure to write down exactly what you are told. That way, you can compare several accounts of the same problem and look for similarities. Consider, for example, the following accounts of a reported bug in a system:

1. "I started by trying to setup a Favorites list. I first went to the home page. Then, I selected Favorites from the menu on the right. I scrolled down to the third entry and pressed Enter on the keyboard. Then I moved to the fourth entry on the submenu and clicked it with the mouse. Finally, I entered my name as Irving, clicked on OK, and the program crashed."

2. "I went to the programming menu and selected the New menu option. I then clicked on the Create Object menu entry and entered the name HouseObject for the object name field. Then I clicked OK, and the program crashed."

3. "I selected the New Project menu option, and then clicked on the icon that looks like a Gear. I entered the name Rudolph for the project name, and then clicked OK. The program crashed."

    - After all, they appear to relate to three different sections of the program.
      Most people, when reporting bugs, focus on what they think was the important step in the process.
    - Filter out the unimportant information and see what each case really has in common.
      First, each user clicked on the OK button to finalize the process, and the program crashed.
    - It might seem likely, therefore, that the program OK handler contains a fatal flaw.
    - A bit of experimentation will show whether this is the case or not.
    - When we examine the statements of the witnesses, we notice that each was working with a menu.
    - We can see that each person used both the keyboard and the mouse to select from the menus before clicking the OK button.

"I selected Favorites from the menu on the right. I scrolled down to the third entry and pressed Enter on the keyboard. Then I moved to the fourth entry on the submenu and

© Copyright Virtual University of Pakistan

clicked it with the mouse."

"... selected the New menu option. I then clicked on the Create Object menu entry ..."

"I selected the New Project menu option, and then clicked on the icon that looks ..."

- We can infer, therefore, that the user in this case used both the keyboard and mouse.
  The next logical place to look is in the menu handler to see whether it deals with mouse and keyboard entries differently.
- The final clue is in the third entry, where the user did not select anything from the menu with the mouse, but instead clicked on an icon.
- While looking at the code, I would try to see what happens when the program deals with a combination of keyboard and menu entry.
- It is likely, given the user discussion, that you will find the root of the problem in this area.
- After finding such a common bug, it is likely that the fix will repair a whole lot of problems at once
- This is a debugger's dream.

Good clues, Easy Bugs

Get A Stack Trace

In the debugging process a stack trace is a very useful tool.
Following stack trace information may help in debugging process.

- Source line numbers in stack trace is the single, most useful piece of debugging information.
- After that, values of arguments are important
  - Are the values improbable (zero, very large, negative, character strings with non-alphabetic characters?
- Debuggers can be used to display values of local or global variables.
  - These give additional information about what went wrong.

Non-reproducible bugs

- Bugs that won't "stand still" (almost random) are the most difficult to deal with.
- Randomness itself, however, is information.
- Are all variables initialized? (random data in variables could affect output).
- Does bug disappeared when debugging code is inserted? Memory allocation (malloc) problems are probably a culprit.
- Is the crash site far away from anything that could be wrong?
- Check for dangling pointers.

Example

```
char *msg(int n, char *s)
{
  char buf[100];
```


© Copyright Virtual University of Pakistan

```
   sprintf(buf, "error %d: %s\n",
            n, s);
   return buf;
}
...
p = msg(20, "Output values");
...
q = msg(30, "Input values");
...
printf("%s\n",p);
```

Lecture No. 45

Summary

© Copyright Virtual University of Pakistan