Software Engineering – 1 (CS504)
Lecture Notes

Delivered by
Dr. Fakhar Lodhi

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

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

Lecture No. 1
Introduction to Software Engineering
An Introduction to Software Construction Techniques for Industrial Strength Software

Introduction
Software engineering is an interesting subject. In order to understand this subject we will need to look at a n

What is Software?
When we write a program for computer we named it as software. But software is not just a program; many th
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 rela

So the software is not just the code written in Cobol, Java, Fortran or C++. It also includes the data and all th

Why is it important?
Undoubtedly software is playing a vital role in all the field of life these days. We can see many software appl

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 ana
Modern scientific investigation and engineering problem solving: Scientific investigations and engineering pr

calculations and data analysis. The accuracy of these analyses is also very important in scientific application
Games: We see many computer games these days that interests people of all ages. All these games are dri
Embedded systems: We see many kinds of gadgets being employed in our daily used things, like small micr

Similarly in many other fields like education, office automation, Internet applications etc, software is being us

Engineering
Before moving on to software engineering lets first discuss something about engineering itself. If you survey
“The process of productive use of scientific knowledge is called engineering.”
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 eng

”This is the process of utilizing our knowledge of computer science in effective production of software system
Difference between Software and Other Systems
Now lets talk something about how a software system is different from any other systems. For example, how

mechanical or electrical engineering. Lets look at some of the non-software systems like TV, Car or an Elect

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 wo

Source of Inherent Complexity of Software
Here the subject is again the same that how software systems are different from other systems. Have you ev

they are making a new system altogether. In other words they are making changes in their systems in many

Therefore one of the major reasons of complexity in software is due to its basic nature that the software pass
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 a

Let’s imagine a person who use to live in a village and who have constructed a hut for him to live. Definitely

In early 60s software had suffered from the similar kind of problem to which we call Software Crisis. Techniq

In most of the cases that software which was tried to be build using those old tools and techniques were not
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 no
As a result of these problems a conference were held in 1960 in which the term software crisis was introduce

More Complex Software Applications
This conception is also very common these days. People think that if one knows how to code then that’s suff

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). IEE

“The application of a systematic, disciplined, quantifiable approach to the development, operation, and maint

Before explaining this definition lets first look at another definition of Software Engineering given by Ian Som

“All aspects of software production’ Software engineering is not just concerned with the technical processes

These definitions make it clear that Software Engineering is not just about writing code.

Software Engineering
Software Engineering is the set of processes and tools to develop software. Software Engineering is the com

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 engine

These days object-oriented programming is widely being used. If programming languages will not support ob

Well-Engineered Software
Let's talk something about what is well-engineered software. Well-engineered software is one that has the fo
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 conform
Software has very close relationship with economics. Whenever we talk about engineering systems we alwa

The major challenges for a software engineer is that he has to build software within limited time and budget i

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

The Balancing Act!
Software Engineering is actually the balancing act. You have to balance many things like cost, user friendlin
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
In some software the efficiency is more important and desirable. For example if we talk about a cruise missil

Therefore software development is a process of balancing among different characteristics of software descri

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 disso
The law of diminishing act describes the same phenomenon. Similar is the case with software engineering. V

Software Background
Caper Jones a renounced practitioner and researcher in the filed of Software Engineering, had made immen
He divided software related activities into about twenty-five different categories They have analyzed around

such a categorization. But here to cut down the discussion we will only describe nine of them that are listed l

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 ter

Fred Brook is a renowned software engineer; he wrote a great book related to software engineering named ‘

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 t

So, according to Fred Brook, in the eye of an unsophisticated manager software is like a giant. Sometimes it

unfortunately magic is not a reality. We do not have any magic to defeat this giant. There is only one solution

Therefore, Software Engineering is nothing but a disciplined and systematic approach to software developm

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.

Lecture No. 02 Introduction to Software Development
Software Development
We have seen in our previous discussion that software engineering is nothing but a disciplined approach to
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
Requirement Gathering
Design Development
Coding
Testing
Management
Management activities are kind of umbrella activities that are used to smoothly and successfully perform the
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 construct

Figure1: Development activities
A Software Engineering Framework
Any Engineering approach must be founded on organizational commitment to quality. That means the softwa
Quality Focus: As we have said earlier, the given framework is based on the organizational commitment to q
Processes: The processes are set of key process areas (KPAs) for effectively manage and deliver quality so

Methods: Methods provide the technical “how-to’s” to carryout these tasks. There could be more than one te

Tools: Tools provide automated or semi-automated support for software processes, methods, and quality co

Figure 2: A Software Engineering Framework

Software Development Loop
Lets now look at software engineering activities from a different perspective. Software development activities
Problem Definition: In this stage we determine what is the problem against which we are going to develop so

Technical Development: In this stage we try to find the solution of the problem on technical grounds and bas

Solution Integration: If there are already developed system(s) available with which our new system has to int

Status Quo: After going through the previous three stages successfully, when we actually deployed the new

After getting new requirements we perform all the steps in the software development loop again. The softwa

Figure 3: Software Development Loop

Software Construction
Here once again look at the construction activities of the software from a different perspective. This section p
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

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

Definition: Here we actually realize or automate the vision developed in first phase. Here we determine what

Development: Here we determine, what should be the design of the system, how will it be implemented and
Maintenance: This is very important phase of software development. Here we control the change in system,

Figure 4: Software Engineering Phases

Maintenance
Correction, adaptation, enhancement
For most large, long lifetime software systems, maintenance cost normally exceeds development cost by fac

Boehm (1975) quotes a pathological case where the development cost of an avionics system was $30 per li

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

Lecture No. 3
Requirement Engineering
Requirement Engineering
We recall from our previous discussion that software development is not simply coding – it is a multi-activity

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

These questions force us to look at the software development process from different angles and require diffe

Requirement engineering mainly deals with the definition phase of the system. Requirement engineering is t

Software Requirements Definitions
Before talking about the requirement process in general and discussing different tools and techniques used f

Jones defines software requirements as a statement of needs by a user that triggers the development of a p

Alan Davis defines software requirements as a user need or necessary feature, function, or attribute of a sys

According to Ian Summerville, requirements are a specification of what should be implemented. They are de

IEEE defines software requirements as:

A condition or capability needed by user to solve a problem or achieve an objective.
A condition or capability that must be met or possessed by a system or system component to satisfy a contra
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 softwa

Importance of Requirements
Many of the problems encountered in SW development are attributed to shortcoming in requirement gatherin

Fred Brooks in his classical book on software engineering and project management “The Mythical Man Mont

“The hardest single part of building a software system is deciding precisely what to build. No other part of the

Let us try to understand this with the help of an analogy of a house. If we are at an advanced stage of buildin

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 su
Role of Requirements

Software requirements document plays the central role in the entire software development process. To start

Once these requirements have been finalized, the construction process starts. During this phase the softwar

On the other hand, the project manager would need this document to monitor and track the progress of the p

The following diagram depicts this central role of the software requirement document in the entire developme

Lecture No. 4
Requirement Engineering-2
Some Risks from Inadequate Requirement Process
From the above discussion, it should be clear that the requirements play the most significant role in the softw

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

Creeping user requirements contribute to overruns and degrade product quality. Requirement creep is one c

It basically means identifying and adding new requirements to the list at some advanced stages of the softwa

Ambiguous requirements lead to ill-spent time and rework.

Ambiguity means that two different readers of the same document interpret the requirement differently. Amb
Let us consider the following requirement statement:

The operator identity consists of the operator name and password; the password consists of six digits. It sho

This is an example of ambiguous requirement as it is not clear what is meant by “it” in the second sentence a

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 impor

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 co

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 valu
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 c

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

Business Requirements:

These are used to state the high-level business objective of the organization or customer requesting the sys

User Requirements:

User requirements add further detail to the business requirements. They are called user requirements becau

Functional Requirements:

The next level of detail comes in the form of what is called functional requirements. They bring-in the system

Non-Functional Requirements

In the last section we defined a software requirement as a document that describes all the services provided and performance attributes. These also include regulations, standards, and contracts to which the product m

Non-functional requirement play a significant role in the development of the system. If not captured properly,

While writing these requirements, it must always be kept in mind that all functional requirements must derive

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 ab
In the next step we need to describe what tasks must be included to accomplish the above-mentioned busin

After documenting the user's perspective in the form of user requirements, the system's perspective: what is
Finally, a non-functional requirement of the system could require that it must be integrated into the existing w

Stakeholders

As mentioned earlier, in order to develop a good requirement document, it is imperative to involve all kinds o

A requirement engineer should be cognizant of the fact that stakeholders have a tendency to state requirem

The role of stakeholders cannot be overemphasized. A study of over 8300 projects revealed that the top two

The following diagram shows the role of different stakeholders in the setting the system requirements.

Requirement■Statement■and■Requirement■Specification Documents

Different levels of software requirements are documented in different documents. The two main documents p

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

Necessary -Each requirement should document something that the customer really need or something that i

Prioritized - An implementation priority must be assigned to each requirement, feature or use case to indicat

Unambiguous - All readers of a requirement statement should arrive at a single, consistent interpretation of i

Verifiable – User should be able to devise a small number of tests or use other verification approaches, such

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 requirem

Let us try to understand this with the help of some examples. The following set of (non-functional) requireme

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 l

Following is another set of (functional) requirements that conflicted with one another:

System must monitor all temperatures in a chemical reactor.

System should only monitor and log temperatures below -200 C and above 4000 C. In this case the two requ

Modifiable - One must be able to revise the Software Requirement Specification when necessary and mainta

Traceable - One should be able to link each requirement to its origin and to the design elements, source cod

Mixed level of Abstraction
It is important to recognize that all requirements in a requirement document are stated at a uniform level of a

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 iden

Lecture No. 5

Relationship■of■Several■components■of■Software Requirements
The following figure depicts the relationship between different documents produced during the requirement e

Business Requirements

Business requirements collected from multiple sources might conflict. For example, consider a kiosk product

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 i
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 retaile
before the kiosk's software requirements are detailed.
You can also use the business requirements to set implementation priorities for use cases and their associat

The Vision Statement

The vision statement should reflect a balanced view that will satisfy the need of diverse customers. It can be

Chemical Tracking System

The chemical tracking system will allow scientists to request containers of chemicals to be supplied by chem

Assumptions and Dependencies

All assumptions that were made when conceiving the project have to be recorded. For example, the manage

Scope

Project scope defines the concept and range of the proposed solution, and limitations identify certain capabi

Scope and Initial Release

The major features that will be included in the initial release of the project should be summarized. Describe t
Requirements need to be prioritized and a release schedule should be made.

The Context Diagram

The scope description establishes the boundary between the system we are developing and everything else

Following is a context diagram of the chemical tracking system.

Use Cases and Customer-Developer Relationship

It has been mentioned earlier on, excellent software products are the result of a well- executed design based
It is important to recognize that a software engineer is typically not hired to solve a computer science proble

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 ex

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. Ea

A use case model represents a use case view of the system – how the system is going to be used. In this ca

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, t

With the help of this diagram, it can be clearly seen that a Book Borrower can reserve a book, borrow a book

Creating a Use Case Model
Creating a use case model is an iterative activity. The iteration starts with the identification of actors. In the n

concurrent and a use case model will evolve slowly from these activities. This activity stops when no new us

Relationship among Use Cases
The UML allows us to extend and reuse already defined use cases by defining the relationship among them.

Let is try to understand these two concepts with the help of the following diagrams. In the case of the first dia

The second diagram demonstrates the concept of reuse by extending already existing use cases. In this cas

This diagram also demonstrates that many different actors can use one use case. Additionally, the actors de

The concept of reusability can also be used in the case of actors. In this case, new classes of actors may be

Credit Card Validation System
Perform Card Transaction

Customer
Process Customer Bill
Retail Institution

Reconcile Transactions

Individual Customer
Corporate Customer

Manage Customer Acct

Sponsoring Financial

Extended User
Institution

In this case two new classes, Individual Customer and Corporate Customer, are being created by extending

Elaborated Use Cases
After the derivation of the use case model, each use is elaborated by adding detail of interaction between th

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:

User

Use Case Name: Delete Information

Priority: 3

Actors: User

Summary: Deleting information allows the user to permanently remove information from the system. Deleting

Preconditions: Information was previously saved to the system and a user needs to permanently delete the i

Post-Conditions: The information is no longer available anywhere in the system.

Uses: Record Transactions, Cancel Action

Extends: None

Normal Course of Events:
The use case starts when the user wants to delete an entire set of information such as a user, commission p
The user selects the set of information that he/she would like to delete and directs the system to delete the i
The system responds by asking the user to confirm deleting the information.
The user confirms deletion.
Alternative Path: Cancel Action
A system responds by deleting the information and notifying the user that the information was deleted from t
Uses: Record Transaction
This use case ends.

Alternative Path - The user does not confirm Deletion
If the user does not confirm deletion, the information does not delete.
Uses: Cancel Action

Exceptions:
The system will not allow a user to delete information that is being used in the system.
The system will not allow a user to delete another user that has subordinates.

Assumptions:
Deleting information covers a permanent deletion of an entire set of data such as a commission plan, user, g
Deleted information is not retained in the system.
A user can only delete information that has not been used in the system.
Alternative Ways of Documenting the Use Case
Many people and organizations prefer to document the steps of interaction between the use and the system

It is a matter of personal and organizational preference. The important thing is to write the use case in prope
Activity Diagrams
Activity diagrams give a pictorial description of the use case. It is similar to a flow chart and shows a flow fro

Delete Information

Choose Object to Delete

Initiate Deletion

[Delete Not Allowed]

[Delete Allowed]
[Cancel Delete]

[Confirm Delete]

Record Action

■■Object Deleted

Limitations of Use Cases
Use cases alone are not sufficient. There are kinds of requirements (mostly non- functional) that need to be

The non-functional requirements are also not documented in the use cases. As examples of those, consider

Usability
Color blind people should not have any difficulty in using the system – color coding should take care of comm
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,

Lecture No. 7

Source and sink analysis
Once requirements are documented using any of these analysis models, an independent verification is need
Source
A stakeholder describes requirements (needs, constraints) to be included as system functionality. These can
Sink
Sink is the consumer of certain information. It is that entity which provides a logical end to a business proces

In source and sink analysis the analyst determines all the sources of requirements and where do these requ
In a similar manner, at times we gather data in our application that is not used anywhere. So the question re

For example, we are having certain inputs (sources) to a process against which we do not know about the co

A stakeholder may have required the development team to develop certain report for his use. It means we ar
A requirement statement that describe the report but do not list down its sources, will be an incomplete state

Process Models
Domain Models
During requirements analysis phase, different models are developed to express requirements of the system.
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 drive
An important difference between software and another engineering discipline is that the software engineer h
The following subsections discuss some of these techniques.
Logical System Models

System models are techniques used to understand user needs and software engineer use these techniques
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 vie

As opposed to flow charts, there are parallel activities in this diagram which are further elaborated by specify
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 consulta
After verifying an IPD patient, system will create a visit and allocate him a room or a bed etc. If system canno

System displays information about the expenses of the required service to the patient so that he is informed
Some advance payment is also received against the required service and this amount is adjusted in the final
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 fill
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 ot
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 d
It may contain information on processes that need not be automated.

Lecture No.8
State Transition Diagrams

State transition diagrams (STDs) are another technique to document domain knowledge. In many cases, inf

Following is an example of a use of STD to document the life cycle of a trouble ticket (this example has beer

A Trouble report and its life cycle – and introduction
From time to time all systems, including communications networks, develop problems or malfunctions referre
At the time of a trouble, a network may have been inter-working with another network to provide a service, a

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
rouble Status attribute is defined which qualifies the state (finer granularity) e.g. cleared awaiting customer v
.
Following is a description of states of a trouble report.

Queued
A trouble report is in a queued state when it has been instantiated but the trouble resolution process has not

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 Res

Deferred
This state indicates that corrective action to resolve the trouble has been postponed. This can occur when th

Cleared
A trouble report is moved by the agent to the “cleared” state when it determines that the trouble has been re

Closed
This state indicates that the trouble resolution process is complete. Upon closure, the trouble report attribute

Disabled
A “disabled” value is exhibited when a trouble report’s information cannot be updated due to local conditions
The following figure shows the STD for a trouble ticket. This diagram depicts the movement of a trouble ticke

Arranging information in tabular form
Sometimes it is better and more convenient to arrange information in a tabular form. This makes it easier for
External Inputs
An external input (EI) is an elementary process that processes data or control information that comes from o
External Outputs
An external output (EO) is an elementary process that sends data or control information outside the applicati
External Inquiry
An external inquiry (EQ) is an elementary process that sends data or control information outside the applicat
It is difficult to understand these definitions and one has to read them a number of times to understand what

Now the same information is presented in the tabular form as follows:

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

do. Identification of EQ is simple - in this case the only thing a function does is present information to the use

Hence by putting and organizing the information in the form of a table, we have not only made it simple to ur

Let us look at another example. This time the information is taken from the Income Tax Ordinance of Pakista

If the taxable income is less than Rs. 60,000, there will be no income tax. If the income exceeds Rs. 60,000

The same information can be organized in the form of a table, making it more readable and easier to use.

Once the information has been organized in the tabular form, in many cases it can be simply stored and map

kind of a rule is simply reduced to a table or dictionary lookup. This reduces the complexity of the domain an

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 o
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
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.

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.

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

Processes
Reconcile account balance
Deposit funds into an account
Pay a bill
Withdraw funds from an account
External agents
Bank
Creditor
Employer
Other income sources
Data stores
Monthly Account statement
Bank accounts
Account transactions
Description:
First we shall discuss ‘withdraw funds from an account’ process. In this process, information about the accou

bank accounts of the employees. Similarly, income received through other income sources is also received a

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 re
A number of parallel activities are shown in this diagram where no specific sequence among these activities
All the previous models that we studied like business process models, state transition diagrams, are used to
However, in data flow models, we represent only those processes which we need to automate as they involv
For example, we may consider a mail desk in an office that receives mail and just forwards it to their respect
In nutshell, processes that just move or transfer data (do not perform any processing on that data), should n
Taking the same example, if we modify the scenario such that a mail desk clerk receives the mail, notes it do

Lecture No. 9

Typical Processes
Now we shall discuss processes which are typically modeled using data flow diagrams. These processes tra
Processes that take inputs and perform certain computations. For example, Calculate Commission is a proc
Processes which are involved in some sort of decision-making. For example, in a point of sales application a
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.
Processes that sort data and present the results to users. For example, we pass an array of arbitrary numbe
Processes that trigger some other function/process
For example, monthly billing that a utility company like WAPDA, PTCL generates. This is a trigger that invok

Actions performed on the stored data. These are called CRUD operations and described in the next subsect
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.
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 perfor

analyst to know each bit of all the processes of the system from the very beginning. Keeping the complexity
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
Detailed Data Flow diagrams
Once context of a system has been captured using context level diagram, the analyst would expand his activ
In level two of data flow model, instead of refining the previous levels further, we take one process from the l
This process may continue to any level of details as the analyst can conveniently captures. Where diagram a
It should be noted here that the number of external agents and their inputs to the system and the outputs tha

which a system is modeled using data flow modeling technique where three levels of abstraction have been
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 f

In order to see detail processes involved in Patient Monitoring System, a level 1 data flow diagram will have

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
It should be noted here that this level 1 diagram is a further refinement of level 0 diagram such that the unde
A further refinement of this model is also possible if we expand any of these three processes to capture furth
Central Monitoring System – Level 2 Data Flow Diagram

In the above level 2 data flow diagram, Patient's data is sent to the Unpack Signs process which unpacks it a
By going through this example, the reader would have learnt how data flow modeling technique helps in und

Common Mistakes in Data Flow Diagrams
In the following data flow diagram, an accounting system has been described. Three processes are given Ge
If you look at the arrows going inside each of these processes and coming out of them, you will observe som
In fact, here we can apply the source and sink analysis that we studied in the last lectures. What does the so

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 r
The Freeze Member Account process that does not have any input is an example of a requirement whose so

needed is an account number and the time period for which the statement is required. If we analyze the inpu
In the above mentioned example, it is evident that by applying the source and sink analysis we determined a
In the following subsection, we shall describe actions which are not only mistakes but illegal too for the data
Illegal Data Flows
Directly Communicating External Agents
Following diagram depicts a scenario in which one external entity is directly communicating with another ext

There must be an intermediate process which should transform data received from one external entity and th

External■Agent■updating■information■in■a■Data Store

As we explained in the above case, a transform/process is needed between communicating entities. This is

Therefore, a process should be inserted between the interacting entities (external agent, data store) that sho

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 fro

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 illeg
So, the correct method is again to use a data transform/process between the two data stores. It should retrie

GUI Sketches
Lecture No. 10
Prototyping and GUI Design

Adding user interface details in the SRS is controversial. The opponents of this argue that by adding GUI de
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

The following GUI implements the delete component use case that we discussed in use case section. The G
The next GUI implements the scenario when user has clicked over the arrow and a few component types are

Following GUI depicts the scenario when user selects a particular plan ‘Plan 3’ and clicks on the ‘Delete’ but

The next GUI, another dialog box is shown in which user is getting another message from the system. It say

The user then selects ‘Plan 2’ and deletes it. System confirms the user and upon confirmation, deletes ‘Plan

After deleting ‘Plan 2’, it displays the message that Plan 2 has been permanently deleted. Whereas, ‘Plan 2’

However, it should be noted that, all the above GUIs presented two major mistakes about the GUIs. First, if a
The following GUI displays what this GUI should have displayed ideally. As, user can only delete plans 1 and

In the above example, it is evident that if requirements are partially generated a number of changes have to

Prototype
Prototyping is yet another technique that can be used to reduce customer dissatisfaction at the requirement

A prototype is not the real product. It is rather just a real looking mock-up of what would be eventually delive

Lecture No. 11
Software Design

Introduction
Recalling our discussion of software construction process, once the requirements of a software system have
It includes modeling of the data structures and entities, the physical and logical partitioning of the system into

Managing Complexity of a Software System
A complex system that works is invariably found to have evolved from a simple system that worked. The stru

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 a

A complex system may be divided into smaller pieces of lesser complexity called modules. This is the classi

Software Design Process
Software design is not a sequential process. Design of a software system evolves through a number of iterat

A activities performed at this stage include design of the software architecture by showing the division of sys

Software Design Strategies
Software design process revolves around decomposing of the system into smaller and simpler units and the

In the functional design, the structure of the system revolves around functions. The entire system is abstract

The object-oriented design takes a different approach. In this case the system is decomposed into a set of o

The object-oriented approach has gained popularity over the structured design approach during the last dec

Software Design Qualities
A software design can be looked at from different angles and different parameters can be used to measure a
Maintainable Design
Since, in general, maintenance contributes towards a major share of the overall software cost, the objective
In order to make a design that is maintainable, it should be understandable and the changes should be local

Lecture No. 12

Coupling and Cohesion
Coupling is a measure of independence of a module or component. Loose coupling means that different sys

Strong cohesion implies that all parts of a component should have a close logical relationship with each othe

A component should implement a single concept or a single logical entity. All the parts of a component shou

Coupling and cohesion are contrasting concepts but are indirectly related to each other. Cohesion is an inter

A good example of a system with a very high cohesion and very less (almost nil) coupling is the electric subs

Modules with high cohesion and low coupling can be treated and analyzed as black boxes. This approach th

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 dep
modules can be identified easily. In this case intra component linkages are stronger while inter component li

Example of Coupling
The modules that interact with each other through message passing have low coupling while those who inter

High Coupling

Low Coupling

In order to understand this concept, let us consider the following example. In this example, we have a class v

```
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 follo

```
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

```
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 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 c

Example of Cohesion
As mentioned earlier, strong cohesion implies that all parts of a component should have a close logical relati

A class will be cohesive if most of the methods defined in a class use most of the data members most of the

Class X1
f1■f1
f2■f2

f3
f4■Class X2

f3 f4

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
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 speci

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 addLineIter
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 setCustometPhon
void setCustomerFax(int cFax) private:
int customerId;
string customerName; int customerPhone; int customerFax;
};
```

Abstraction and Encapsulation

Abstractions is a technique in which we construct a model of an entity based upon its essential characteristic

Engineers of all fields, including computer science, have been practicing abstraction for mastering complexit

```
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 c

```
viod swap(int &x, int &y)
{
int temp; temp = x; x = y;
y = temp;
}


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 find

Function Oriented versus Object Oriented Design

Let us now try to understand the difference between object-oriented and function oriented (or action oriented

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

This difference is elaborated with the help of the following diagram:

Functions

Data

In this diagram, the ovals depict the function while rectangles/squares depict data. Since a function contains

Object oriented approach solves this problem by putting the relevant data and functionality together at one p

Let us assume that the circles represent sets of functions and rectangles represent data that these function u

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, softwa

A complex system that works is invariably found to have evolved from a simple system that worked. The stru

Some of the key advantages which make the object-oriented technology significantly attractive than other te

Clarity and understandability of the system, as object-oriented approach is closer to the working of human co

Reusability of code resulting from low inter-dependence among objects, and provision of generalization and

Reduced effort in maintenance and enhancement, resulting from inheritance, encapsulation, low coupling, a

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 differen

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

This difference is elaborated with the help of the following diagram:

Functions

Data

In this diagram, the ovals depict the function while rectangles/squares depict data. Since a function contains

In the case of OO design since data and function are put together in one class, hence, in case of a change,

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 exhibi

The state of an object encompasses all of the properties of the object and their current values. A property is

Behavior is how an object acts and reacts in terms of its state changes and message passing. The behavior

The structure and behavior of similar objects are defined in their common class. A class represents an abstr
Classification
The most important and critical stage in the OOA and OOD is the appropriate classification of objects into gr

Data-Driven
head, tail, body, leg
Behavior-Driven
walk, run, eat
Responsibility-Driven
carry things, communicate, maintain its living system

Here, we can take a data-driven, behaviour driven, or responsibility driven perspective and will categorize th

The Object Model
The elements of object oriented design collectively are called the Object Model. The object model encompas

Abstraction is an extremely powerful technique for dealing with complexity. Unable to master the entirety of a

Abstraction and encapsulation are complementary concepts. Abstraction provides the outside view to the cli
Relationship Among Objects
The object model presents a static view of the system and illustrates how different objects collaborate with o

Inheritance defines a “kind of” hierarchy among classes. By inheritance, we specify generalization/specializa

In an association relationship, when object A “uses” object B, then A may send messages to B. The relations

The aggregation relationship defines part-of structure among objects. When object A is part of the state of ob

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

As mentioned earlier, aggregation is the “part-whole” or “a-part-of” relationship in which objects representing
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 b
Coupling and Linkages
As mentioned earlier, aggregation implies a much tighter coupling than association. In case of aggregation, t
Ownership and visibility
Another way of differentiating among the two is to look at them from the ownership and sharing point of view

In other words, in case of aggregation, the whole owns its parts and the part becomes a private property of t
Database persistence
From a database perspective, when an object is persisted or stored in the database, all of its components (a

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 occ

Static Model
Identify classes (i.e. attributes and methods are defined)
Specify class hierarchy
Identify object-to-object relationships
Model the object behavior
Dynamic Model
Scenario Diagrams

Object Oriented Design
OOD transforms the analysis model into design model that serves as a blueprint for software construction. C

The subsystem layer. Contains a representation of each of the subsystems that enable the software to achie
The class and object layer. Contains the class hierarchies that enable the system to be created using genera
The message layer. Contains the details that enable each object to communicate with its collaborators. This
The responsibility layer. Contains the data structures and algorithmic design for all attributes and operations

Translating the analysis model into a design model during object design

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

Part of speech Model component■Example

proper noun improper noun doing verb being verb having verb adjective adjective phrase
instance class/type/role operation classification composition
attribute value or class association
operation
Mehdi Hassan student, teacher buy
is a horse, is a book fan has wings
this ball is green
the customer with children
the customer who bought the kite

Once all the model components have been identified, we will eliminate the redundant or irrelevant componer
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 f
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)

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

We will continue with technique to identify all the constituent components of the model and derive our object

Lecture No. 15
The Notation
Many different notations are used for documenting the object oriented design. Most popular of these include

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
Select Objects – who am I?
We have used an approach that divides the objects into different categories to make it easier to find them ar
Select actors
Actors are people and organizations that take part in the system under consideration. Examples of actors ar
Select Participants
A participant is a role that each actor plays in the system under consideration. Examples of participants are:
Select Places
Places are where things come to rest or places that contain other objects. Examples of places are: airport, a
Select Transactions
Transactions are the “events” that must be remembered through time. These are entries that must be mainta

information. Examples of transactions are: agreement, assignment, authorization, contract, delivery, deposit
Select Container Objects
Containers are objects that hold other objects. Note the similarity of definition between container and places
Select Tangible things
Take a “walk” through the system and select “tangible” things around you used in the problem domain. Thes

While selecting objects, the following considerations should be kept in mind for a simpler (and better) object

Every object that you put in your object model should have some responsibility or role to play in the problem

Avoid having controller objects because controllers usually end up with functionality that’s better done by oth
In large systems several objects are likely to have similar or even identical responsibilities. Look for such obj

Use meaningful class names, names that describe objects in that class. Try to use names from the domain v

Lecture No. 17
Identify Structures
A structure is a manner of organization which expresses a semantically strong organization within the proble
Identify Gen-Spec Structures (Hierarchy)
Consider each class that you have identified as a specialization and then look for its generalization and vice
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

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
For each object include the attributes that come to mind when you first think about the object. The criteria fo

For actors consider name, address, phone.
For participants consider number, date and time, password, authorization level.
For place/location consider number, name, address (perhaps latitude, longitude, altitude).
For transaction consider number, date, time, status.
For line item consider quantity, status.
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 attrib

An attribute that varies over time, e.g., price of an item, should be replaced by an additional class with an eff
An attribute that may have a number of values should be replaced by a new class and an object connection.
Replace “yes/no” type attributes with “status” type attributes for flexibility.
If there are classes with common attributes and generalization-specialization makes good sense, then add a

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 collaborat

For an actor, include an object connect to its participants (association).
For a participant, include an object connection to its actor (already established) and its transactions (associa
For a location, include object connections to objects that it can hold (association), to its part objects (aggrega
For transactions, include object connections to its participants (already established), its line items (aggregati
For a transaction line item, include object connections to its transaction (already established), its item (assoc
For an item, include object connections to transaction line item (already established), a companion “item des
For a composite object, include object connections to its “part” object (aggregation).
For all objects (including all of the above) select connecting objects to which the object under consideration s
Define Services - What I do?
The third and last step in establishing each object’s responsibility is to define what services does each objec

Software objects do things that the system is responsible to do with regard to that object. By putting the serv
Why does the system need this object any way?
What useful questions can it answer?
What useful action can it perform?
What this object can do, based upon what it knows?
What this object can do, based upon whom it knows?

What calculations can it do?
What ongoing monitoring could it do?
What calculations across a collection could it make (letting each worker do its part)?
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:

For an actor, consider: calculate for me, rate me, is <value>, rank participants, calculate over participants.
For a participant, consider: calculate for me, rate me, is <value>, rank transactions, calculate over transactio
For a place, consider: calculate for me, rate me, is <value>, rank transactions, calculate over contents, calcu

For a Transaction, consider: calculate for me, rate me, is <value>, how many, how much, rank transaction li
For a line item, consider: calculate for me, rate me.
For an item, consider: calculate for me, rate me, is <value>, how many, how much, rank, calculate over spec

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

Why ?
balance the cash in the drawer with the amount recorded by the point-of-sale system.

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
Identify system features
Be certain to include features that cover the following
log important information
conduct business
analyze business results
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

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 ther
Customer
customer. You must have a way to know about customer objects; otherwise it should not be put in the doma
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:

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

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.

■

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

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 Perform

Places:
store
Attributes:■name
Services:■get item for UPC, get cashier for number

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 o

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

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
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

Who I know? register, cashier, store, sales

Object Model Diagram for Connie’s Convenience Store

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 dor

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 sequ

The Sequence Diagram
Let us first look at Sequence Diagrams. These diagrams illustrate how objects interacts with each other and

The Notation
Following diagram illustrates the notation used for drawing sequence diagrams.

Lifeline■Message

The boxes denote objects (or classes), the solid lines depict messages being sent from one object to the oth

These concepts are further elaborated with the help of the following sequence diagram.
X-Axis (objects)

borrow(book)

message

ok = mayBorrow()

[ok] borrow(member)

condition
Life Line

setTaken(member)

Object

Activation box

As shown above, in a sequence diagram, objects (and classes) are arranged on the X-Axis (horizontally) wh

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

An interaction between two objects is performed as a message sent from one object to another. It is most of

actual message sent through some communication mechanism, either over the network or internally on a co

If object obj1 sends a message to another object obj2 an association must exist between those two objects.

A message is represented by an arrow between the life lines of two objects. Self calls are also allowed. Thes
Arguments and control information (conditions, iteration) may also be included. It is preferred to use a brief t

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, asyn

Synchronous Messages

Synchronous messages are “call events” and are denoted by the full arrow. They represent nested flow of c

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 mes
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 cal
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 obj

Sequence diagrams and logical complexity
It is important to judiciously use the sequence diagrams where they actually add value. The golden principle

Collaboration diagrams
Collaboration diagrams can also be used to depict the dynamic behaviour of a system. They show how obje

Collaboration diagrams have basically two types of components: objects and messages. Objects exchange m

The following diagrams illustrate the use of collaboration diagrams.

1 : Add professor (Professor)

1 : set course info

: Registrar
3 : add course

4 : <<create>>

Comparing sequence & collaboration diagrams

Sequence diagrams are best to see the flow of time. On the other hand, static object connections are best re

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 t

You may also recall our earlier discussion of coupling and cohesion. It can be easy to see that OO design yi

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, Ac

If we encapsulate the three objects into one Room object as shown below, then the Heat Flow Regulator will

This happened because in the first case intelligence is distributed while in the second case it is encapsulated

By doing that we reduce coupling even further because now we have made Room more cohesive by putting

That is, we can reduce the coupling of a system by minimizing the number of messages in the protocol of a c

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 b

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

Lecture No. 22

Software and System Architecture

Introduction
When building a house, the architect, the general contractor, the electrician, the plumber, the interior designe

The design process for identifying the sub-systems making up a system and the framework for sub-system c

The study of software architecture is in large part a study of software structure that began in 1968 when Eds

David Parnas pressed this line of observation with his contributions concerning information-hiding modules,

A program family is a set of programs (not all of which necessarily have been or will ever be constructed) for

Parnas argued that early design decisions should be ones that will most likely remain constant across memb

produce. In the context of this discussion, an early design decision is the adoption of a particular architecture

All of the work in the field of software architecture may be seen as evolving towards a paradigm of software

Before talking about the software architecture in detail, let us first look at a few of its definitions.

What is Software Architecture?

What do we mean by software architecture? Unfortunately, there is yet no single universally accepted definit

Before looking at the definitions for the software architecture, it is important to understand how a software sy

According to UML 1.3, a system is a collection of connected units that are organized to accomplish a specific

Let us now look at some of the software architecture definitions from some of the most influential writers and

Software Architecture Definitions UML 1.3:
Architecture is the organizational structure of a system. An architecture can be recursively decomposed into

Bass, Clements, and Kazman. Software Architecture in Practice, Addison-Wesley 1997:

'The software architecture of a program or computing system is the structure or structures of the system, wh

By "externally visible" properties, we are referring to those assumptions other components can make of a co

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 p

IEEE Glossary

Architectural design: The process of defining a collection of hardware and software components and their int

Shaw and Garlan

The architecture of a system defines that system in terms of computational components and interactions am

Each of these definitions of software architecture, though seemingly different, emphasizes certain structural

One can thus conclude from these definitions that an architectural design is an early stage of the system des

activities and It includes the high-level design of modular components, their relationships and organization, a

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 f

Why is architecture important and why is it worthwhile to invest in the development of a architecture? Fundar

Mutual communication. Software architecture represents a common high-level abstraction of the system that

Each stakeholder of a software system (customer, user, project manager, coder, tester, and so on) is concer

Early design decisions. Software architecture represents the embodiment of the earliest set of design decisio

An implementation exhibits an architecture if it conforms to the structural design decisions described by the a

Resource allocation decisions also constrain implementation. These decisions may be invisible to implement

aspects of algorithm design or the intricacies of the programming language, but they are the ones responsib

Not only does architecture prescribe the structure of the system being developed, but it also engraves that st

A side effect of establishing the work breakdown structure is to effectively freeze the software architecture, a

Thus, once the architecture’s module structure has been agreed on, it becomes almost impossible for manag

Is it possible to tell that the appropriate architectural decisions have been made (i.e., if the system will exhibi

Reusable abstraction of a system. Software architecture embodies a relatively small, intellectually graspable

In an architecture-based development of a product line, the architecture is in fact the sum of the early design

(or a family of closely related architectures) that will serve all envisioned members of the product line by mak

A family-wide design solution may not be optimal for all systems derived from it, but the quality gained and la

Architectural Attributes

Software architecture must address the non-functional as well as the functional requirements of the software

Performance
Performance can be enhanced by localising operations to minimise sub- system communication. That is, try
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 a
Maintainability
Maintainability is directly related with simplicity. Therefore, maintainability can be increased by using fine-gra

Architectural design process

Just like any other design activity, design of software architecture is a creative and iterative process. This inv

System structuring

System structuring is concerned with decomposing the system into interacting sub-systems. The system is d

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.

Lecture No. 23
8.6 Architectural Views
Software architecture defines the high level structure of the software by putting together a number of archite
Software architecture = {Elements, Forms, Rationale}
That is, a software architecture is a set of elements which have a certain form. These elements are further d
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
This model is shown in the following diagram.

Logical View

End-user
Functionality

Use Cas

Process View
System integrators Performance Scalability Throughput
Deployment View
System engineering System topology Delivery, installation Communication

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 view

The Functional View comprises of various different functions provided by the system, key system abstraction

The Development View documents the different files and directories in the system and users of this view incl

The components of the Code View include classes, objects, procedures, functions, subsystems, layers, and

Concurrency View intends to document different parallel processes and threads in the system. Its main focus

The Physical View depicts the physical organization of this system and how this system will be physically de

Finally, there are Scenarios. Scenarios are basically use cases which describe the sequences of responsibil

This is summarized in the following table:

What Are Views Used For?

Views are an engineering tool to help achieve desired system qualities. Each view provides an engineering h

Structures can also be used to document how the current system was developed and how future developme

Although one often thinks about a system's structure in terms of functionality, there are other system propert

Hierarchical Views

Every view is potentially hierarchical, e.g. functional view could contain sub-functions. Similarly development

Architectural views are related to each other in complicated ways. One has to choose the views that are use

Architectural styles

The architectural model of a system may conform to a generic architectural model or style. An awareness of

Each style describes a system category that encompasses:

a set of components (e.g., a database, computational modules) that perform a function required by a system
a set of connectors that enable "communication, coordination and cooperation" among components,
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 kn
Lecture No. 24

Architectural models
Like analysis models, many different kinds of architectural models are developed during the architectural des

Architectural Styles

Architectural design may be based upon a certain pattern of model. These different patterns are also called a

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:

Shared data is held in a central database or repository and may be accessed by all sub- systems
Each sub-system maintains its own database and passes data explicitly to other sub- systems

When large amounts of data are to be shared, the repository model of sharing is most commonly used. This
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 c

Disadvantages

Repository model suffers from a number of disadvantages. First of all, sub- systems must agree on a reposit

Client-server model

Client server model tries to distribute data and processing. This is a shift from main- frame based application

The client-server model is a distributed system model which shows how data and processing is distributed a

services. These clients and servers are connected through a network which allows clients to access servers

The following diagram depicts a general client-server organization.

Server
Client

Client/Server Software Components

A typical client-server architecture based system is composed of a number of different components. These i

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 ne
Database servers
In this case, client sends requests, such as SQL queries, to the database server, the server processes the re
Transaction servers
In this configuration, client sends requests that invokes remote procedures on the server side, server execut
Groupware servers
Groupware servers provide set of applications that enable communication among clients using text, images,

Client-server characteristics

Advantages
The main advantage of the client-server architecture is that it makes effective use of networked systems. Ins

Disadvantages
The main disadvantage of this model is that there is no standard way of sharing data, so sub-systems may u

Representative Client/Server Configurations

The client-server model can have many different configurations. In the following sections, we look at some o

Thin Client Model

This model was initially used to migrate legacy systems to client server architectures. In this case the legacy

Fat Client Model

With advent of cheaper and more powerful hardware, people thought of using the processing power of client

Lecture No. 25

Zero Install
As discussed earlier, fat-client architecture posed major challenges in terms of installation and maintenance

N-Tier architecture

N-tier architecture stems from the struggle to find a middle ground between the fat- client architecture and th

Three-tier Architecture

In this architecture, each application architecture layers (presentation, application, database) may run on sep

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 h

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

If the dataflow has only a single sequence of processes with no alternative or parallel paths, then it is called

Batch Sequential

Layered Architecture

As the name suggests, a layered architecture has many different layers. One typical example of a layered ar

User Interface Layer Application Layer

Utility Layer

Core Layer

Reference architectures

Reference architecture is not a physical architecture. It is only a reference for defining protocols and designi

7
6
5
4
3
2
1

OSI reference model

Partitioning the Architecture

Partitioning of architecture is an important concept. What we basically want to do is distribute the responsibil

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 ar

Vertical partitioning divides the application from a decision making perspective. The architecture is partitione

Top Partition
Bottom Partition Workers

Analyzing Architecture design

In a given system, the required characteristics may conflict. Trade-offs seek optimal combinations of propert

Collect scenarios.
Elicit requirements, constraints, and environment description.
Describe the architectural styles/patterns that have been chosen to address the scenarios and requirements
Evaluate quality attributes by considered each attribute in isolation.
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.

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 env

Even though Alexander was talking about patterns in buildings and towns, what he says is true about object-
Design Patterns defined
“Description of communicating objects and classes that are customized to solve a general design in a particu

Patterns are devices that allow programs to share knowledge about their design. In our daily programming, v

Essay writing is usually done in a fairly well defined form, and so is documenting design patterns. The gener

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 lat

Language[Alex77] and A Timeless Way of Building [Alex79] which, in addition to giving examples, described

The pattern movement became very quiet until 1987 when patterns appeared again at an OOPSLA conferen

The concept of design patterns is not new as we can find a number of similar pursuits in the history of progra
Anti-patterns is another concept that corresponds to common mistakes in analysis and design. These are id
With this introduction, we now describe the format that has been adopted in GoF book for describing various
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.

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

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
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
Dependence/publish-subscribe mechanism in programming language
Smalltalk being the first pure Object Oriented language in which observer pattern was used in implementing
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 t
Description
This can be used when multiple displays of state are needed.
Consequences
Optimizations to enhance display performance are impractical.

Example implementation of Observer PatternObject Model
Many graphical user interface toolkits separate the presentational aspects of the user interface from the und

Structure
Participants

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 accessi
Singleton pattern should be used when controlling the total number of instances that would be created for a
Singleton pattern should be used when the sole instance should be extensible by sub classing and clients sh
Structure

Participants
Singleton

Defines an instance operation that lets clients access its unique instance. Instance is a class operation (that
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 init
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 c
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

You would use façade pattern when there are many dependencies between clients and the implementation c
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 co

Structure

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.

Lecture No. 28
Good Programming Practices and Guidelines

Maintainable Code
As we have discussed earlier, in most cases, maintainability is the most desirable quality of a software artifa

Self Documenting Code
From a maintenance perspective, what we need is what is called self documenting code. A self documenting

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 inclu

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.

Identifier Names

Identifier names also play a significant role in enhancing the readability of a program. The names should be

```
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 c

```
if (AllocFlag == 0)
```

The situation has improved a little bit but the semantics of the condition are still not very clear as the meanin

```
If (AllocFlag == NEW_NUMBER)
```

We have improved the quality of the code by replacing the number 0 with a named constant NEW_NUMBER

Coding Style Guide

Consistency plays a very important role in making it self-documenting. A consistently written code is easier t

This coding style guide emphasizes on C++ and Java but the concepts are applicable to other languages as

Naming Conventions

Hungarian Notation was first discussed by Charles Simonyi of Microsoft. It is a variable naming convention t

Bicapitalization or camel case (frequently written CamelCase) is the practice of writing compound words or p

In our style guide, we will be using a naming convention where Hungarian Notation is mixed with CamelCase

General Naming Conventions

General naming conventions for Java and C++
Names representing types must be nouns and written in mixed case starting with upper case.
Line, FilePrefix

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

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 meth
int■getMaxIterations()//■NOT:■MAX_ITERATIONS■=■25
{
return■25;
}
Names representing methods and functions should be verbs and written in mixed case starting with lower ca
getName(), computeTotalWidth()

Names representing template types in C++ should be a single uppercase letter.
template<class■T>■... template<class C, class D> ...

Global variables in C++ should always be referred to by using the :: operator.
::mainWindow.open() , ::applicationContext.getName()

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

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 variabl

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;

All names should be written in English.
fileName;■// NOT:■filNavn

Variables with a large scope should have long names, variables with a small scope can have short names. S

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.

Specific Naming Conventions for Java and C++
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);

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

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;

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
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 computation
The term initialize can be used where an object or a concept is established.
printer.initializeFontSet();

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), matrixEle
A list in this context is the compound data type that can be traversed backwards, forwards, etc. (typically a V
n prefix should be used for variables representing a number of objects.
nPoints, nLines

The notation is taken from mathematics where it is an established convention for indicating a number of obje

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
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. Variabl

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/

Reduce complexity by symmetry.

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
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 abbreviation
HypertextMarkupLanguage instead of■html CentralProcessingUnit■instead of■cpu PriceEarningRatio■inst

Negated boolean variable names must be avoided.

The problem arise when the logical NOT operator is used and double negative arises. It is not immediately a
Functions (methods returning an object) should be named after what they return and procedures (void meth
Line *line■//NOT Line *pLine; or Line *lineptr; etc

Lecture No. 29
File handling tips for Java and C++

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.

Classes should be declared in individual header files with the file name matching the class name. Secondary

```
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 implem

Special characters like TAB and page break must be avoided. These characters are bound to cause proble

Include Files and Include Statements for Java and C++

Header files must include a construction that prevents multiple inclusion. The convention is an all uppercase
#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 t

Classes and Interfaces
Class and Interface declarations should be organized in the following manner:
Class/Interface documentation.
class or interface statement.
Class (static) variables in the order public, protected, package (no access modifier), private.
Instance variables in the order public, protected, package (no access modifier),
private.
Constructors.
Methods (no specific order).

Statements in Java and C++
Types
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 intentior

Types that are local to one file only can be declared inside that file.

The parts of a class must be sorted public, protected and private. All sections must be identified explicitly. No

Variables
Variables should be initialized where they are declared and they should be declared in the smallest scope po

Variables must never have dual meaning. This enhances readability by ensuring all concepts are represente
Class variables should never be declared public. The concept of information hiding and encapsulation is viol

Related variables of the same type can be declared in a common statement. Unrelated variables should not
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
Variables should be kept alive for as short a time as possible. Keeping the operations on a variable within a

Global variables should not be used. Variables should be declared only within scope of their use. Same is re

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

Loop structures

Only loop control statements must be included in the for() construction.

Loop variables should be initialized immediately before the loop.
boolean done = false; // NOT:■boolean done = false; while (!done) {■//■■:
:■//■while (!done) {
}■//■:
}

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 numbe

The use of break and continue in loops should be avoided. These statements should only be used if they pro

The form for (;;) should be used for empty loops.
for (;;) {■// NOT:■while (true) {
:■//■:
}■//■}

This form is better than the functionally equivalent while (true) since this implies a test against true, which is

Conditionals
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 = elementN
if (isFinished || isRepeatedEntry) {
:
}

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 {
:
}

The conditional should be put on a separate line.
if (isDone)■// NOT: if (isDone) doCleanup(); doCleanup();

Executable statements in conditionals must be avoided.
file = openFile (fileName, "w"); // NOT:■if ((file = openFile (fileName, "w")) != null) {
if (file != null) {■//■:
:■//■}
}

Miscellaneous
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.

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

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 mathe

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.

goto in C++ should not be used. Goto statements violates the idea of structured code. Only in some very few to be less readable.

Lecture No. 30
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

As Fowler puts it, comments should not be used as deodorants. Tricky code should not be commented but r

If, however, there is a need to write comments for whatever reason, the following guidelines should be obser

All comments should be written in English. In an international environment English is the preferred language
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 po

Comments should be indented relative to their position in the code.

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 deep

Natural form for expression

Expression should be written as if they are written as comments or spoken out aloud. Conditional expressior

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

if (x & MASK == BITS)

This causes problems because == operator has higher precedence than & operator. Hence, MASK and BITS

if ((x & MASK) == BITS)

Following is another example of the use of parentheses which makes the code easier to understand and her

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 t
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 co

*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 i

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

Shortcuts and cryptic code

Sometimes the programmers, in their creative excitement, try to write very concise code by using shortcuts a

Let us start with a very simple shortcut, often used by programmers. Assume that we have the following stat

```
x *= a;
```

For some reason the code was later modified to:

```
x *= a + b;
```

This seemingly harmless change is actually a little cryptic and causes confusion. The problem lies with the s

Let us now look at a more complex example. What is the following code doing? subkey = subkey >> (bitoff –
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
subkey = subkey >> (bitoff & 0x7);
It is quite evident that the second piece of code is much follow to read than the first one.
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 re

```
a = a/4;
```

A piece of code similar to the following can be found in many data structures books and is part of circular imp

```
bool Queue::add(int n)
{
int k = (rear+1) % MAX_SIZE; if (front == k)
return false;

else {



}
}

rear = k; queue[rear] = n; return true;




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 obviou

In an experiment, the students were asked to implement double-ended queue in which pointer could move in

It is always much better to state the logic explicitly. Also, counting is also much easier to understand and cod

```
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;
```

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 th

```
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 co

```
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;
}
```

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 constar

The difference would be evident if we look at the code segment below that achieves the same purpose as th

```
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, ' ');


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 segmen

```
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 c

```
flag = false; str = NULL; name[i] = '\0'; x = 0.0;
i = 0;
```

Lecture No. 32

Clarity through modularity

As mentioned earlier, abstraction and encapsulation are two important tools that can help in managing and r

```
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 perfor

```
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 abs

Reusability is one of the prime reasons to make functions but is not the only reason. Modularity is of equal c

```
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){




}
} while (i < j); temp = a[left]; a[left] = a[j]; a[j] = temp;
```

```
temp = a[i]; a[i] = 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 st

```
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;
}
```

Short circuiting || and &&
The logical and operator, &&, and logical or operators, ||, are special due to the C/C++ short circuiting rule, i.

Short-circuiting is a very useful tool. It can be used where one boolean expression can be placed first to “gua

Let us look at the following code segment taken from a commercially developed software for a large internat

```
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

```
while (ptr != NULL && ptr->data < myData){
// do something here
}
```

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 para

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 e

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 pa

```
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
```

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
```

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 w
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. Combin

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 initia

Guidelines
If the following guidelines are observed, one can avoid hazards caused by side effects.
never use “,” except for declaration
if you are initializing a variable at the time of declaration, do not declare another variable in the same statem
never use multiple assignments in the same statement
Be very careful when you use functions with side effects – functions that change the values of the parameter
Try to avoid functions that change the value of some parameters and return some value at the same time.

Performance
In many cases, performance and maintainability are at odds with one another. When planning for performan

As an example, consider the following. In this example a function isspam is profiled by calling 10000 times. T

The profiling revealed that most of time was spent in strchr and strncmp and both of these were called from s

When a small set (a couple of functions) of functions which use each other is so overwhelmingly the bottlene

use a better algorithm
rewrite the whole set

In this particular case strstr was rewritten and profiled again. It was and found out that although it was much

The algorithm was rewritten and restructured again by eliminating strstr, strchr, and strncmp and used memo

strlen now went from over two million calls to 652.

Many details of the execution can be discovered by examining the numbers. The trick is to concentrate on h

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 tota

for (j = i; j < maxField; j++) clear(j);

This reduced the overall execution time by half.

Lecture No. 34
Portability
Many applications need to be ported on to many different platforms. As we have seen, it is pretty hard to wri

Stick to the standard

Use ANSI/ISO standard C++
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

int i, j, k;
…
j = 20000;
k = 30000;

i = j + k;

// works if int is 4 bytes
// what will happen if int is 2 bytes?

Order of Evaluation
As mentioned earlier during the discussion of side effects, order of evaluation varies from one implementatio

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 sig

Interestingly, Java has introduced a new operator to handle this issue. >> is used for for arithmetic shift and

Byte Order and Data Exchange

The order in which bytes of one word are stored is hardware dependent. For example in Intel architecture th

Alignment

The C/C++ language does not define the alignment of items within structures, classes, an unions. data may

```
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 typeca

Bit Fields

Bit fields allow the packing of data in a structure. This is especially useful when memory or data storage is at

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 fun
} 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 th

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

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 betwee
Bit fields therefore should not be used.

Lecture No. 35
Exception handling
Exception handling is a powerful technique that separates error-handling code from normal code. It also prov

The idea is to raise some error flag every time something goes wrong. There is a system that is always on th

try {
■...
■...
■throw Exception()
■...
■...
} catch(Exception e)
{
■...
■...
}

One of the most powerful features of exception handling is that an error can be thrown over function bounda

Exceptions and code complexity
A number of invisible execution paths can exist in simple code in a language that allows exceptions. The cor

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:
Different order of evaluating function parameters are ignored.
Failed destructors are ignored
Called functions are considered atomic
for example: e.Title() could throw for several reasons but all that matters for this function is whether e.Title()
To count as different execution paths, an execution path must be made-up of a unique sequence of function

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:

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
if e.Title() != “CEO” and e.Salary() > 10000
both parts of the condition will be evaluated
cout will be performed.
if e.Title() != “CEO” and e.Salary() <= 10000
both parts of the condition will be evaluated
cout will not be performed.

Exceptional Code Paths
The 20 exceptional code path are listed below.

String EvaluateSalaryAnadReturnName(Employee e)

The argument is passed by value, which invokes the copy constructor. This copy operation might throw an e

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 th

if (e.Title() == “CEO” || e.Salary() > 10000) Same as above.

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 typ

if (e.Title() == “CEO” || e.Salary() > 10000) Same as above.

if (e.Title() == “CEO” || e.Salary() > 10000) operator ==() might throw.

if (e.Title() == “CEO” || e.Salary() > 10000) Same as above.

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 t

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.

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.

The Challenge:
Can we make this code exception safe and exception neutral? That is, rewrite it (if needed) so that it works p

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

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 res

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
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 except

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 ha
If the message emitted successfully but an exception occurs in later in the function (for example during the a

message was emitted to cout even though the function failed because of an exception. It should be complete

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 effec

// 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);
```

string copy constructor is invoked because the result is returned by value.
The copy assignment operator is invoked to copy the result into theName
If either copy fails then the function has completed its side-effects (since the message was completely emitte

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 inc

```
// 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 ensu

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-

When such a situation comes with two or more unrelated side-effects which cannot be combined then the be

Summary
Providing the strong exception-safety guarantee often requires you to trade-off performance.
If a function has multiple un-related side-effects, it cannot always be made strongly exception safe. If not, it
Not all functions need to be strongly exception-safe. Both the original code and attempt#1 satisfy the basic g

Lecture No. 36
Software Verification and Validation
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 e
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 expectation
Defect
The second major and a very important concept is Defect. A defect is a variance from a desired product attri
Therefore software defect is that phenomenon in which software deviates from its expected behavior. This is
Software and Defect
Software and defects go side by side in the software development life cycle. According to a famous saying b
Software Testing
With these concepts, we are in a position to define software testing. Software testing is the process of exami

is the process of executing software product on test data and examining its output vis-à- vis the documented
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. Tha
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 absenc
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
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 ve

“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
Limitations of testing
With the help of the following example, we shall see how difficult it may become to discover a defect from a
Example (adapted from Backhouse)
This is a function that compares two strings of characters stored in an array for equality. The tester who has
Inputs and Expected Outputs

Results of testing
The tester runs all the above-mentioned test cases and function returns the same results as expected. But it
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 chara
The above-mentioned defect signifies a clear limitation of the testing process in discovering a defect which is
Testing limitations
In order to prove that a formula or hypothesis is incorrect all you have to do to show only one example in wh
On the other hand, million of examples can be developed to support the hypothesis but this will not prove tha
These examples only help you in coming up with a hypothesis but they are not proves by themselves and th
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

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 co
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 th
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 commo
At the time when these two activities are being performed, merely initial design of the application is complete

We shall explain the testing activities parallel to development activities with the help of the following diagram

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 ger
System comes into testing after development is completed.
Test cases are executed with test data and actual results (application behavior) are compared with the expe
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,
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.
The Developer and Tester

Scenarios missed or misunderstood during development analysis would never be tested correctly because th
The left side column is related to the development, and the right side describes the testing. Development is a
Tester analyzes FS with respect to testing the system whereas; developer analyzes FS with respect to desig
Usefulness of testing
Objective of testing is to discover and fix as many errors as possible before the software is put to use. That i
Therefore, it should be well preserved among the community of developers that testers are essential rather i
Testing and software phases
With the help of the following diagram we shall explain different phases of testing

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 component
Subsystem testing – testing of collection of modules to discover interfacing problems among interacting mod
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
In the following two types of testing activities are discussed.
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 behav
Structural testing (white box)
As opposed to black box testing, in structural or white box testing we look inside the system and evaluate wh
Effective testing
The objective of testing is to discover the maximum number of defects with a minimum number of resources
As, good testing involves much more than just running the program a few times to see whether it works or no
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?

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 C
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 shou
Equivalence partitioning guidelines
Organize your equivalence classes. Write them in some order, use some template, sequence, or group them
Boundary conditions: determine boundary conditions. For example, adding in an empty linked list, adding aft
You should not forget invalid inputs that a user can give to a system. For example, widgets on a GUI, numer
Equivalence partitioning example
In the following example, we shall see how equivalence partitions can be developed for a string matching fu
String matching
Organization
For equivalence partitions, we divide the problem in two obvious categories of equal strings and the other on
Test cases for equivalence partitions Equal
Two equal strings of arbitrary length
All lower case■"cat"■"cat"
All upper case■"CAT"■"CAT"

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
Basis Code Structures
For structural testing it is important to know about basic coding structures. There are four basic coding struct
Flow graph notation
In analysis and design, you have already seen the flow graph notation. This is used to describe flow of data
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 lu

If
Second structural form is the If statement. In the following graph, the first node at the left depicts the if stater

Case
In Case statement, control can take either of several branches (as opposed to only two in If statement.) First

While
A while loop structure consists of a loop guard instruction through which the iteration in the loop is controlled

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 co
Point to note here is the assignment of node numbers to program instructions. As, the corresponding flow gr
So all these combinations are to be tested during white box testing.

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
Coverage
Statement Coverage: In this scheme, statements of the code are tested for a successful test that checks all t
Branch Coverage: In this scheme, all the possible branches of decision structures are tested. Therefore, sec
Path Coverage: In path coverage, all possible paths of a program from input instruction to the output instruct

White Box Testing Example
With the help of the following example, we shall see how many test cases are generated for each type of cov

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 ho

Paths
The following is an analysis of the above-mentioned code and the flow diagram. It determines the number of
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).
o 1-2-4-1-5
o 1-3-4-1-5
N=2: Four possible paths can be traversed.
o 1-2-4-1-2-4-1-5
o 1-2-4-1-3-4-1-5

o 1-3-4-1-2-4-1-5
o 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, 2N p
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
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 exhau

complexity of the code. This will give us a number that corresponds to the total number of test cases that ne

Cyclomatic complexity
The concept of cyclomatic complexity is extremely useful in white box testing when analyzing the relative co
Cyclomatic complexity is a quantitative measure of the logical complexity of a program. It defines number of
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 p
Cyclomatic Complexity of a Sort Procedure
Following is the same bubble sort program that we discussed above. This time we shall calculate its cycloma
Cyclomatic complexity
Number of edges = 8
Number of nodes = 6
C(G) = 8-6+2 = 4
Paths to be tested
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
Infeasible paths
Infeasible path is a path through a program which is never traversed for any input data.
Example

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
Modified code segment
Infeasible paths can be analyzed and fixed.
There are no infeasible paths now!

Lecture No. 40
Unit testing
A software program is made up of units that include procedures, functions, classes etc. The unit testing proc
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
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 s
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 tel
Contract for square root routine
result = squareRoot(argument);
assert (abs (result * result – argument) < epsilon);

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
When you design a module or even a single routine, you should design both its contract and the code to test
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 w
Defect removal efficiency
Is the ability to remove defects from the application. We shall further elaborate this idea with the help of the a

The above table depicts data that was published after analyzing 1500 projects. In these projects, four differe

Inspection and chaotic zone

Requirement Design■Coding■Documentatio
n
Testing
Maintenance

■

In this diagram, a chaotic zone has been defined. In fact, if defects are not discovered and fixed at the appro
If we combine the results of the above two diagrams, it is evident that testing alone does not suffice. We nee
Defect origination
In inspections the emphasis is on early detection and fixing of defects from the program. Following are the p
Requirements
Design
Coding
User documentation
Testing itself can cause defects due to bad fixes
Change requests at the maintenance or initial usage time

It is important to identify defects and fix them as near to their point of origination as possible.

Lecture No. 41
Inspection versus Testing
Inspections and testing are complementary and not opposing verification techniques. Both should be used d

Inspection pre-conditions
A precise specification must be available before inspections. Team members must be familiar with the organ

Inspection checklists.
Checklist of common errors in a program should be developed and used to drive the inspection process. The
Inspection Checklist
Following is an example of an inspection checklist.

In the checklist mentioned above, a number of fault classes have been specified and their corresponding ins

Static analyzers
Static analyzers are software tools for source text processing. They parse the program text and try to discov

Checklist for static analysis

Lecture No. 42

Debugging

Debugging

As Benjamin Franklin said, “in this world, nothing is certain but death and taxes.” If you are in the software d

What is a Bug?
We call them by many names: software defects, software bugs, software problems, and even software “featu
A Brief History of Debugging
It would appear that as long as there have been computers, there have been computer bugs. However, this
After graduating from Vassar in 1928, she went to Yale to receive her master’s degree in mathematics. After
It was during her term with the Mark II that Hopper was credited with coining the term “bug” for a computer p
From that auspicious beginning, computer debugging developed into something of a hit- or-miss procedure f

evolution of debugging came with the advent of command-line debuggers. These simple programs were an a
Once the debugger became a part of the programmer’s arsenal, the software world began to consider other

Importance of Debugging
As we mentioned earlier in this course, one of the prime objectives of software engineering is to develop cos
Problem at AT&T
In the telecommunications industry, loss of service is known as an outage. For most of us, when as outage o
On January 15. 1990, AT&T had a US wide telephone system outage that lasted for nine hours. The cause w
Description of the Problem
The code snippet that caused the outage is illustrated as follows

```
do {
. . .
switch (expression) {
case 0: {
if (some_condition) {


. . .
break;
} else {
. . .
10.■}
. . .
break;
13.■}
14.■. . .
15.■}
. . .
} 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 su
AT&T statement about the Problem
“We believe that the software design, development, and testing processes we use are based on solid, quality

We do not believe we can fault AT&T’s software development process for the 1990 outage, and we have no

Art 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

“code” and when he has to check this code, he can potentially miss out obvious mistakes due to this impress

Program at Bulletin Board Example
Following piece of code was once placed on a bulletin board making an invitation to people to discover the p

```
while (i =0; i < 10; i++) {
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 th

In order to reach such defects, one needs a scientific approach to check and verify the code methodically. B

Lecture No. 43
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 int
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 fo

```
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 n

Example 2
The following code snippet from some old C++ code contains a slightly more insidious bug. It illustrates the p

```
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)
{


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,

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
Coding errors

A coding error is a simple problem in writing the code. IT can be a failure to check error returns, a failure to c
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 in

```
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 cas
Memory over-runs

a memory overrun occurs when you use memory that does not belong to you. This can be caused by overst
Symptoms
Program crashes quite regularly after a given routine is called, that routine should be examined for a possibl
If the routine in question does not appear to have any such problem the most likely cause is that another rou
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

```
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 s
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 surm
Likewise, if you were using graphics software and saw all of the points on the screen, but the last two were u

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 impo

```
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, ass

```
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

you would expect this to happen. The problem, however, is that the next time through the loop, the inner loo
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
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
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
Example
Let’s take a look at one of the most common forms of the pointer error, using a pointer after it has been dele

```
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 problem is, the reuse did not take into account the existing system. This problem is common when reusi
When the second delete occurs at the bottom of func(), the results are unpredictable. At best, the delete func
Boolean bugs
Boolean bugs occur because the mathematical precision of Boolean algebra has virtually nothing to do with

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
For true/false problems, you will usually see some sort of debug output indicating an error in a function, only
Examples
The following code contains a simple example of a function that returns a counter- intuitive Boolean value ar

```
int DoSomeAction(int InputNum)
{
if(InputNum < 1 || InputNum > 10) DoSomeAction = 0;
else
{
PerformTheAction(InputNum); DoSomeAction = NumberOfAction + 1; NumberOfAction = NumberOfAction +
}
return DoSomeAction;
}
```

After looking at the function, ask yourself this – What is the return value in the case of success? What would

```
int Value; int Ret;

Value = 11;


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 investiga

Lecture No. 44
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
Now that you understand what holistic means, how do you apply holistic concepts to debugging the software
As a simple example, consider the case where the program mysteriously crashes at a particular point in the

```
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

Most experienced programmers, managers, or debuggers would say that the problem is not fixed, although f

The debugging process
In normal circumstances, you will have a user description of the problem. This description might have been g

"I started by trying to setup a Favorites list. I first went to the home page. Then, I selected Favorites from the

"I went to the programming menu and selected the New menu option. I then clicked on the Create Object me

"I selected the New Project menu option, and then clicked on the icon that looks like a Gear. I entered the na

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 clickin

"I selected Favorites from the menu on the right. I scrolled down to the third entry and pressed Enter on the

clicked it with the mouse."

"... selected the New menu option. I then clicked on the Create Object menu entry ..." "I selected the New Pr
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 entrie
The final clue is in the third entry, where the user did not select anything from the menu with the mouse, but
While looking at the code, I would try to see what happens when the program deals with a combination of ke
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 de
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
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];


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