- item
- register
- cash drawer
- Tax Category (Descriptive things)

Session

Is it important? It is important in order to evaluate a cashier's performance.

Lecture No. 19

Identify Structures

Identify Gen-Spec Structures

Kinds of stores:

A store is a kind of sales outlet. Perhaps over time, Connie will expand to other kinds of sales outlets. Stores might be specialized into kinds of stores. For now on leave store as it is.

Kinds of sales:

- sales, returns
- only different is that the amount is positive or negative. Is there any other difference?

Prices:

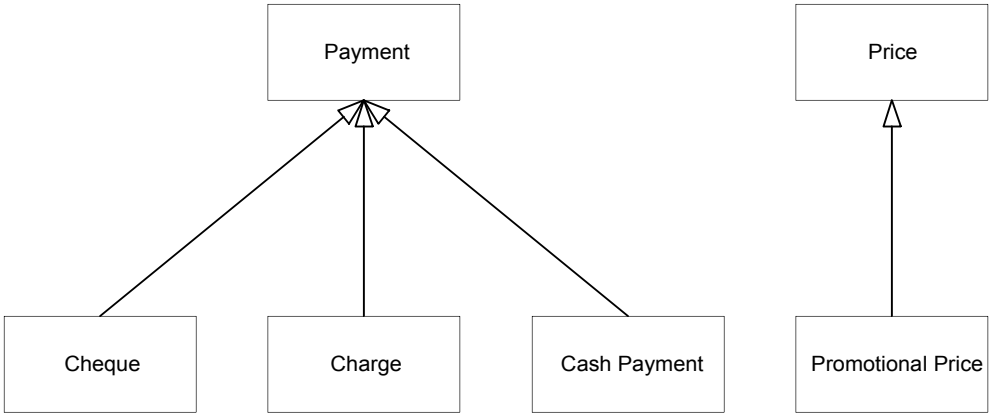
- regular price, and promotional (sales) price

Payment:

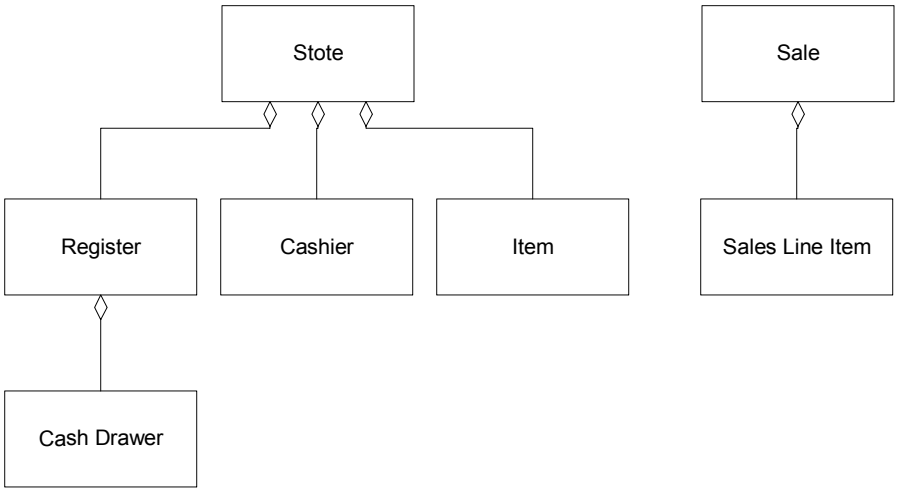
- cash, check, and charge are kind of payments

Identify Whole-Part Structures

- A store as a whole is made up of cashiers, registers, and items.
- A register contains a cash drawer.
- A sale is constituted of sale line items.



Object Hierarchy



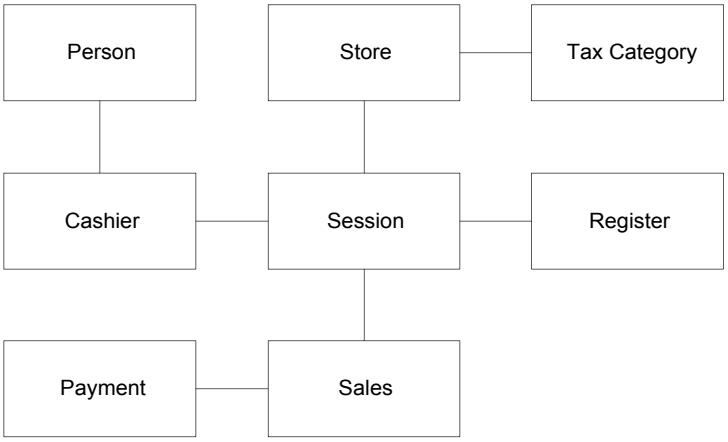
Whole-Part Structures

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 Performance

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 of register)

cash drawer

Attributes: balance, position (open, close), operational state
Services: open
Who I know? register

Tax Category

Attributes: category, rate, effective date
Services: just the basic services - get, add, set - don't show
Who I know? items?

Transactions:

sale

Attributes: date and time
Services: calculate subtotal, calculate total, calculate discount, calculate
tax, commit
Who I Know? session, payment, SLIs

sale line item

Attributes: date and time ?, quantity, tax status (regular, resale, tax-exempt)
Services: calculate sub total
Who I Know? item, sale

sale line item - how do you handle returns and sales

sale - you have control
return - more difficult
- return to a different store
- purchased for a different price
- returns an item no longer in the inventory

return

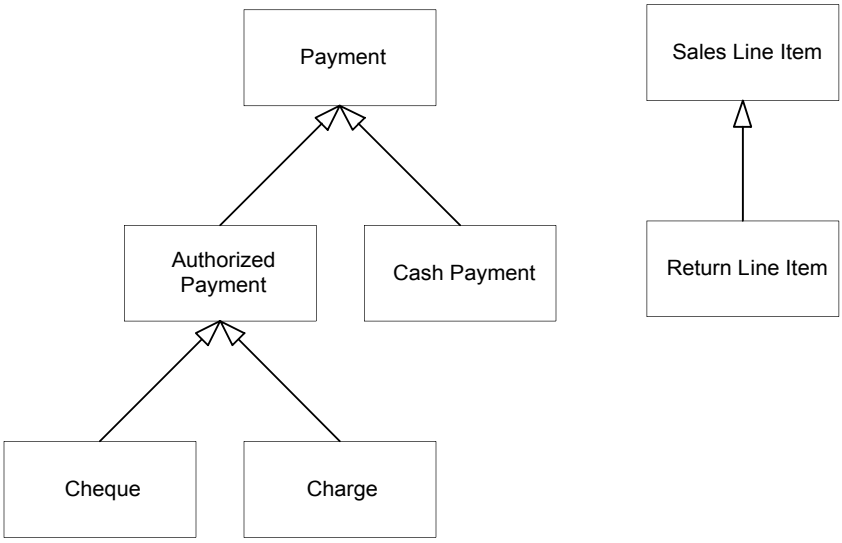
Attributes: return price, reason code, sale date, sale price

Services:
Who I Know?

is it a case for gen-spec, what's same, what's different

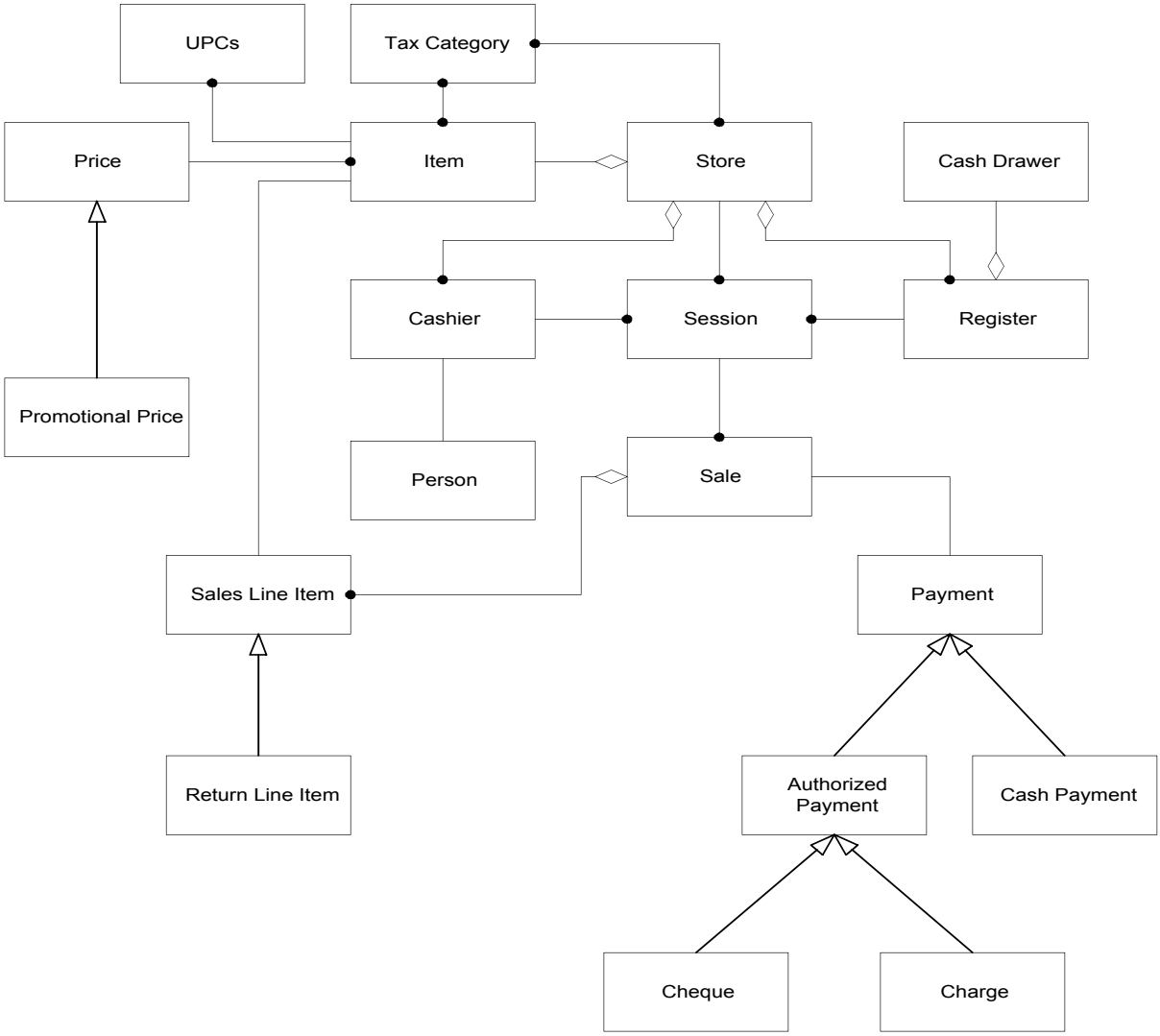
payment - we have types of payments
Attributes:
each payment knows about its
amount paid, cash tendered
a check object knows its
bank, account number, amount tendered, authorization code
a credit object knows about its
card type, card number, expiration date, authorization code
common attributes among check and credit - use gen-spec
hierarchy becomes:
payment
cash payment
authorized payment
check
card

Services:
who I know: sale



session
Attributes: start date, end date, start time, end time
Services: how much money collected over interval, how many sales

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 done in terms of a set of messages exchanged among a set of objects within a context to accomplish a purpose. This is often used to model the way a use case is realized through a sequence of messages between objects.

The purpose of Interaction diagrams is to:

- Model interactions between objects
- Assist in understanding how a system (a use case) actually works
- Verify that a use case description can be supported by the existing classes
- Identify responsibilities/operations and assign them to classes

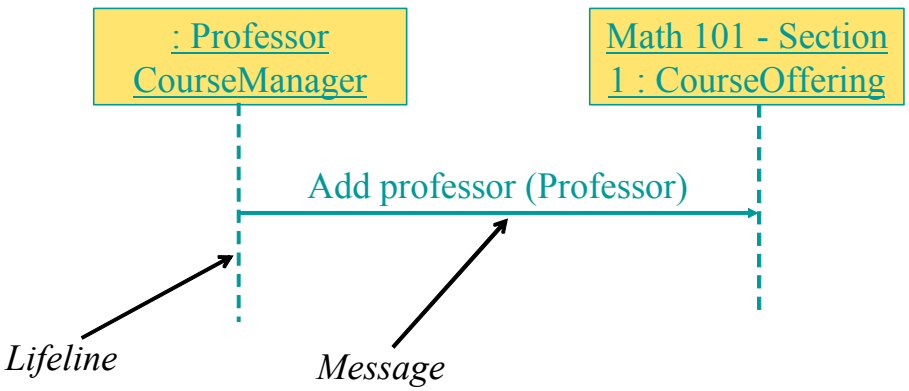
UML provides two different mechanisms to document the dynamic behaviour of the system. These are sequence diagrams which provide a time-based view and *Collaboration Diagrams* which provide an organization-based view of the system's dynamics.

The Sequence Diagram

Let us first look at Sequence Diagrams. These diagrams illustrate how objects interacts with each other and emphasize time ordering of messages by showing object interactions arranged in time sequence. These can be used to model simple sequential flow, branching, iteration, recursion and concurrency. The focus of sequence diagrams is on objects (and classes) and message exchanges among them to carry out the scenarios functionality. The objects are organized in a horizontal line and the events in a vertical time line.

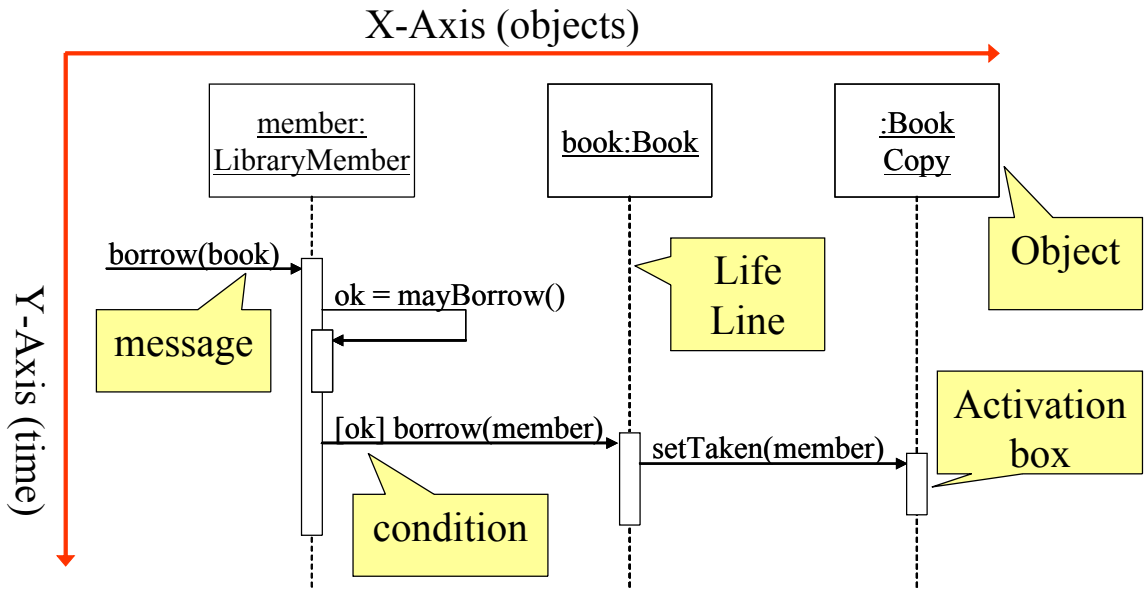
The Notation

Following diagram illustrates the notation used for drawing sequence diagrams.



The boxes denote objects (or classes), the solid lines depict messages being sent from one object to the other in the direction of the arrow, and the dotted lines are called life-lines of objects. The life line represents the object's life during interaction. We will discuss this in more detail later.

These concepts are further elaborated with the help of the following sequence diagram.



As shown above, in a sequence diagram, objects (and classes) are arranged on the X-Axis (horizontally) while time is shown on the Y-Axis (vertically). The boxes on the life-line are called activation boxes and show for how long a particular message will be active, from its start to finish. We can also show if a particular condition needs to occur before a message is invoked simply by putting the condition in a box before the message. For example, object *member:LibraryMember* sends a message to object *book:book* if the value of *ok* is true.

The syntax used for naming objects in a sequence diagram is as follows:

- syntax: [instanceName][:className]
- Name classes consistently with your class diagram (same classes).
- Include instance names when objects are referred to in messages or when several objects of the same type exist in the diagram.

An interaction between two objects is performed as a message sent from one object to another. It is most often implemented by a simple operation call. It can however be an

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

If object obj_1 sends a message to another object obj_2 an association must exist between those two objects. There has to be some kind of structural dependency. It can either be that obj_2 is in the global scope of obj_1, or obj_2 is in the local scope of obj_1 (method argument), or obj_1 and obj_2 are the same object.

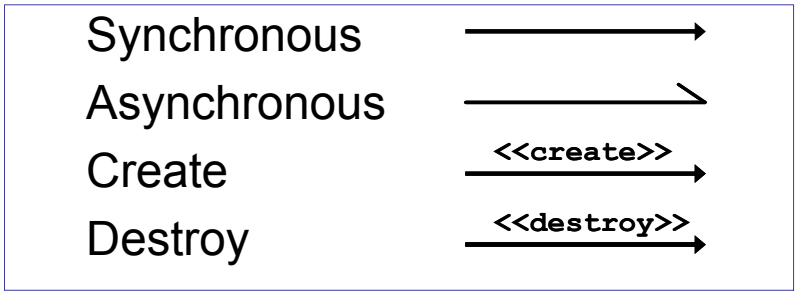
A message is represented by an arrow between the life lines of two objects. Self calls are also allowed. These are the messages that an object sends to itself. This notation allows self calls. In the above example, object *member:LibraryMember* sends itself the *mayBorrow* message. A message is labeled at minimum with the message name.
Arguments and control information (conditions, iteration) may also be included. It is preferred to use a brief textual description whenever an *actor* is the source or the target of a message.

The time required by the receiver object to process the message is denoted by an *activation-box.*

Lecture No. 21

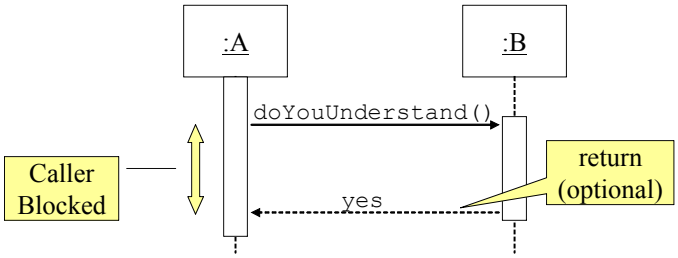
Message Types

Sequence diagrams can depict many different types of messages. These are: synchronous or simple, asynchronous, create, and destroy. The following diagram shows the notation and types of arrows used for these different message types.



Synchronous Messages

Synchronous messages are "call events" and are denoted by the full arrow. They represent nested flow of control which is typically implemented as an operation call. In case of a synchronous message, the caller waits for the called routine to complete its operation before moving forward. That is, the routine that handles the message is completed before the caller resumes execution. Return values can also be optionally indicated using a dashed arrow with a label indicating the return value. This concept is illustrated with the help of the following diagram.



While modeling synchronous messages, the following guidelines should be followed:

- Don't model a return value when it is obvious what is being returned, e.g. getTotal()
- Model a return value only when you need to refer to it elsewhere, e.g. as a parameter passed in another message.
- Prefer modeling return values as part of a method invocation, e.g.
  ok = isValid()

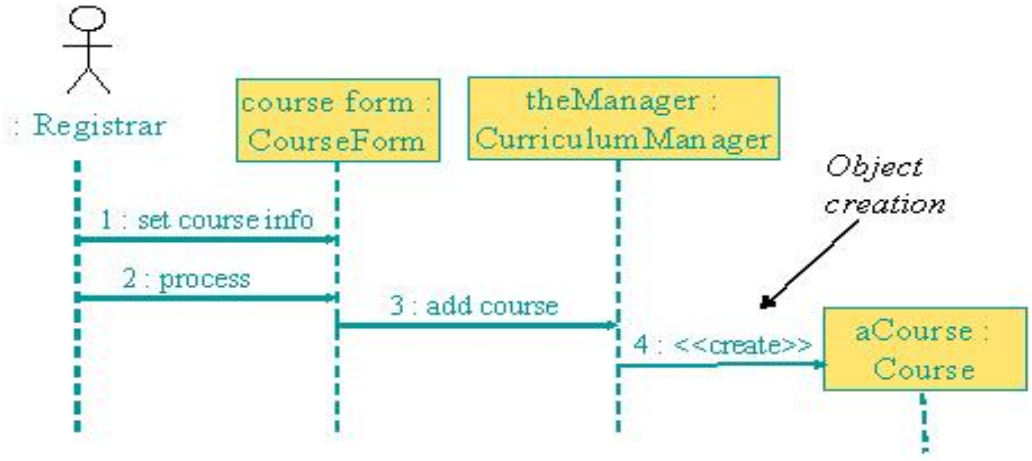
Asynchronous messages

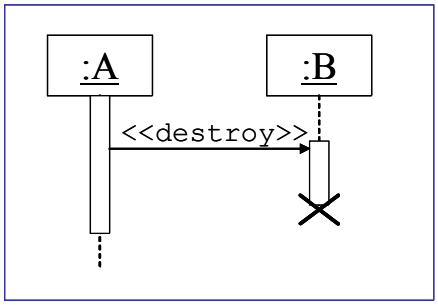
Asynchronous messages are "signals," denoted by a half arrow. They do not block the caller. That is, the caller does not wait for the called routine to finish its operation for continuing its own sequence of activities. This occurs in multi-threaded or multi-processing applications where different execution threads may pass information to one another by sending asynchronous messages to each other. Asynchronous messages typically perform the following actions:

- Create a new thread
- Create a new object
- Communicate with a thread that is already running

Object Creation and Destruction

An object may create another object via a <<create>> message. Similarly an object may destroy another object via a <<destroy>> message. An object may also destroy itself. One should avoid modeling object destruction unless memory management is critical. The following diagrams show object creation and destruction. It is important to note the impact of these activities on respective life lines.





Sequence diagrams and logical complexity

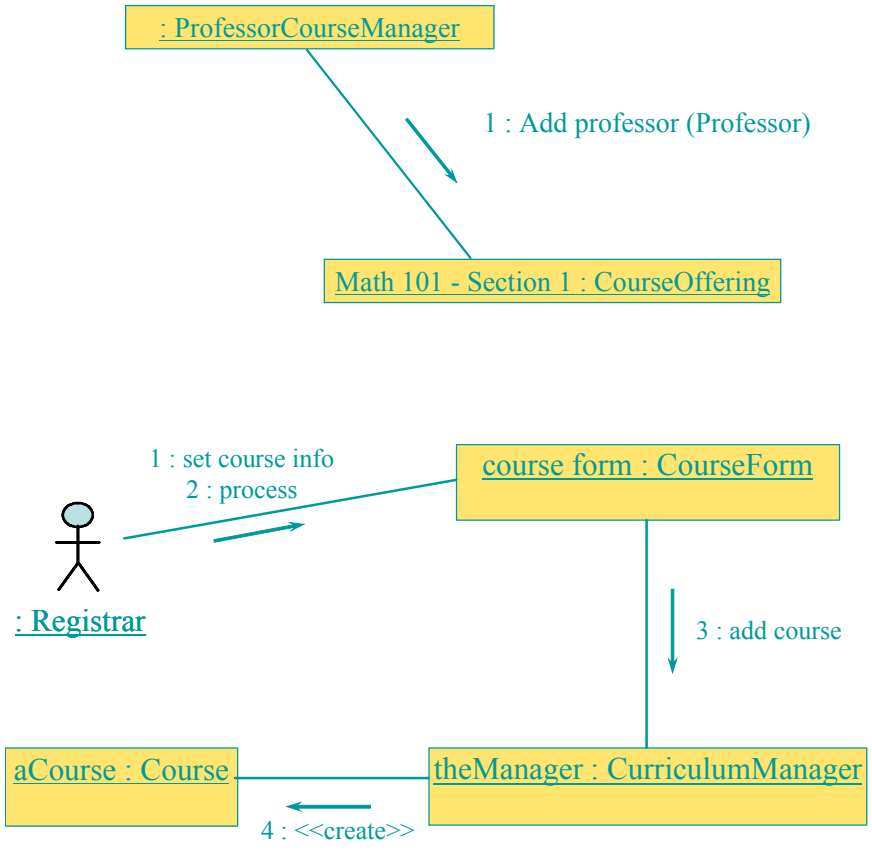
It is important to judiciously use the sequence diagrams where they actually add value. The golden principle is to keep it small and simple. It is important to understand that the diagrams are meant to make things clear. Therefore, in order to keep them simple, special attentions should be paid to the conditional logic. If it is simple then there is no harm in adding it to the diagram. On the other hand if the logic is complex then we should draw separate diagrams like flow charts.

Collaboration diagrams

Collaboration diagrams can also be used to depict the dynamic behaviour of a system. They show how objects interact with respect to organizational units (boundaries!). Since a boundary shapes communication between system and outside world e.g. user interface or other system, collaboration diagrams can be used to show this aspect of the system. The sequence of messages determined by numbering such as 1, 2, 3, 4, … This shows which operation calls which other operation.

Collaboration diagrams have basically two types of components: objects and messages. Objects exchange messages among each-other. Collaboration diagrams can also show synchronous, asynchronous, create, and destroy message using the same notation as used in sequence diagrams. Messages are numbered and can have loops

The following diagrams illustrate the use of collaboration diagrams.



Comparing sequence & collaboration diagrams

Sequence diagrams are best to see the flow of time. On the other hand, static object connections are best represented by collaboration diagrams. Sequence of messages is more difficult to understand in collaboration diagrams than in the case of sequence diagrams. On the other hand, object organization with control flow is best seen through collaboration diagrams. It may be noted that complex control is difficult to express anyway but collaboration diagrams can become very complex very quickly.

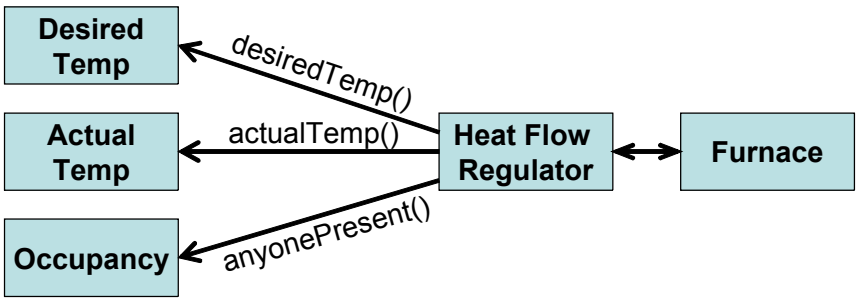
Evaluating the Quality of an Object-Oriented Design

Judging the quality of a design is difficult. We can however look at certain object-oriented design attributes to estimate its quality. The idea is to analyze the basic principle of encapsulation and delegation to judge whether the control is centralized or distributed, hence judging the coupling and cohesion in a design. This will tell us how maintainable a design is.

You may also recall our earlier discussion of coupling and cohesion. It can be easy to see that OO design yield more cohesive and loosely coupled systems.

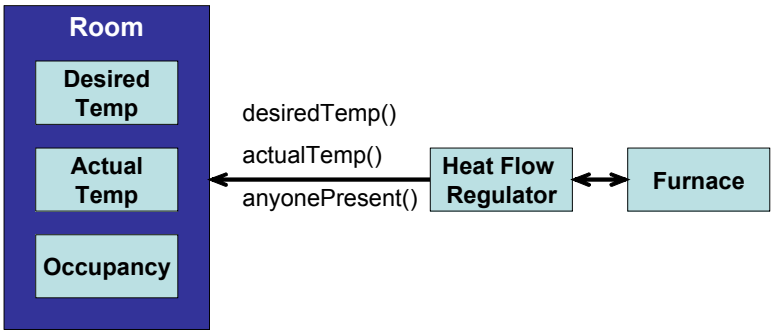
The issue of centralized versus distributed control can be illustrated with the help of the following example.

Consider a heat regulatory system for a room as shown below.

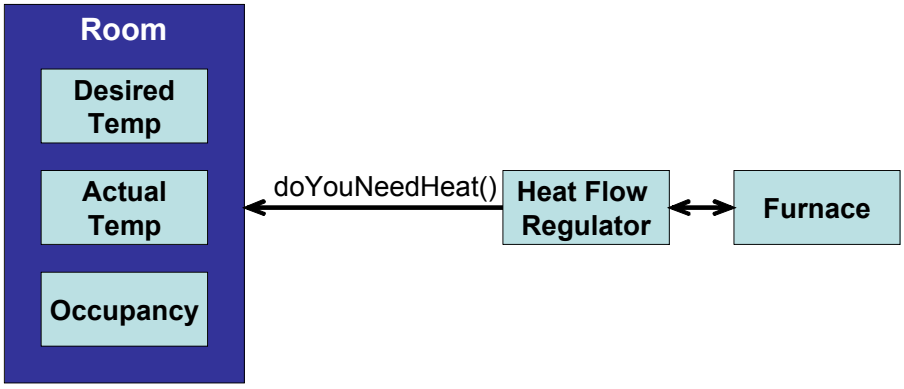


In this case, the room is not encapsulated as one entity and three different objects namely *Desired Temp, Actual Temp,* and *Occupancy* maintain necessary information about a room. In this case the Heat Flow Regulator has to communicate with three different objects.

If we encapsulate the three objects into one Room object as shown below, then the Heat Flow Regulator will need to communicate with one object, hence the overall coupling of the system will be reduced.



This happened because in the first case intelligence is distributed while in the second case it is encapsulated. However, the control is still centralized as the Heat Flow Regulator has the control logic that first analyses the values from different queries and then makes a decision about turning the furnace on of off. We can improve the situation even further by delegating this responsibility to the Room object as shown below.



By doing that we reduce coupling even further because now we have made Room more cohesive by putting the function with the related data and have thus reduced the number and types of messages being sent from the regulator to the room.

That is, we can reduce the coupling of a system by minimizing the number of messages in the protocol of a class. The problem with large public interfaces is that you can never find what you are looking for…smaller public interfaces make a class easier to understand and modify. This can be further elaborated with the help of the following example. Suppose we have two functions defined for setting the desired temperature in the room:

- SetMinimumValue(int aValue)
- SetMaximimumValue(int aValue)

We can reduce the total number of messages in the protocol of this class by consolidation these as shown below, hence reducing the overall complexity of the protocol.

- SetLimits(int minValue, int maxValue)

It is however important to use these kinds of heuristics judiciously and care must be taken so that the scope of the function does not go beyond providing one single operation.

Lecture No. 22

Software and System Architecture

Introduction

When building a house, the architect, the general contractor, the electrician, the plumber, the interior designer, and the landscaper all have different views of the structure. Although these views are pictured differently, all are inherently related: together, they describe the building's architecture. The same is true with software architecture. Architectural design basically establishes the overall structure of a software system.

The design process for identifying the sub-systems making up a system and the framework for sub-system control and communication is *architectural design.* The output of this design process is a description of the *software architecture.*

The study of software architecture is in large part a study of software structure that began in 1968 when Edsger Dijkstra pointed out that it pays to be concerned with how software is partitioned and structured, as opposed to simply programming so as to produce a correct result. Dijkstra was writing about an operating system, and first put forth the notion of a layered structure, in which programs were grouped into layers, and programs in one layer could only communicate with programs in adjoining layers. Dijkstra pointed out the elegant conceptual integrity exhibited by such an organization, with the resulting gains in development and maintenance ease.

David Parnas pressed this line of observation with his contributions concerning information-hiding modules, software structures, and program families.

A program family is a set of programs (not all of which necessarily have been or will ever be constructed) for which it is profitable or useful to consider as a group. This avoids ambiguous concepts such as "similar functionality" that sometimes arise when describing domains. For example, software engineering environments and video games are not usually considered to be in the same domain, although they might be considered members of the same program family in a discussion about tools that help build graphical user interfaces, which both happen to use.[1]

Parnas argued that early design decisions should be ones that will most likely remain constant across members of the program family that one may reasonably expect to

produce. In the context of this discussion, an early design decision is the adoption of a particular architecture. Late design decisions should represent trivially-changeable decisions, such as the values of compile-time or even load-time constants.

All of the work in the field of software architecture may be seen as evolving towards a paradigm of software development based on principles of architecture, and for exactly the same reasons given by Dijkstra and Parnas: Structure is important, and getting the structure right carries benefits.

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

8.2 What is Software Architecture?

What do we mean by software architecture? Unfortunately, there is yet no single universally accepted definition. Nor is there a shortage of proposed definition candidates. The term is interpreted and defined in many different ways. At the essence of all the discussion about software architecture, however, is a focus on reasoning about the *structural* issues of a system. And although architecture is sometimes used to mean a certain architectural style, such as client-server, and sometimes used to refer to a field of study, it is most often used to describe structural aspects of a particular system.

Before looking at the definitions for the software architecture, it is important to understand how a software system is defined. It is important because many definitions of software architecture make a reference to software systems.

According to UML 1.3, a system is a collection of connected units that are organized to accomplish a specific purpose. A system can be described by one or more models, possibly from different viewpoints. IEEE Std. 610.12-1990 defines a system as a collection of components organized to accomplish a specific function or set of functions. That is, a system is defined as an organized set of connected components to accomplish the specified tasks.

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

Software Architecture Definitions

UML 1.3:

> Architecture is the organizational structure of a system. An architecture can be recursively decomposed into parts that interact through interfaces, relationships that connect parts, and constraints for assembling parts. Parts that interact through interfaces include classes, components and subsystems.

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

'The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.

By "externally visible" properties, we are referring to those assumptions other components can make of a component, such as its provided services, performance characteristics, fault handling, shared resource usage, and so on. The intent of this definition is that a software architecture must abstract away some information from the system (otherwise there is no point looking at the architecture, we are simply viewing the entire system) and yet provide enough information to be a basis for analysis, decision making, and hence risk reduction."

Garlan and Perry, guest editorial to the *IEEE Transactions on Software Engineering*, April 1995:

Software architecture is "the structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time."

IEEE Glossary

Architectural design: The process of defining a collection of hardware and software components and their interfaces to establish the framework for the development of a computer system.

Shaw and Garlan

The architecture of a system defines that system in terms of computational components and interactions among those components. Components are such things as clients and servers, databases, filters, and layers in a hierarchical system. Interactions among components at this level of design can be simple and familiar, such as procedure call and shared variable access. But they can also be complex and semantically rich, such as client-server protocols, database accessing protocols, asynchronous event multicast, and piped streams.

Each of these definitions of software architecture, though seemingly different, emphasizes certain structural issues and corresponding ways to describe them. It is important to understand that although they are apparently different, they do not conflict with one another.

One can thus conclude from these definitions that an architectural design is an early stage of the system design process. It represents the link between specification and design processes. It provides an overall abstract view of the solution of a problem by identifying the critical issues and explicitly documenting the design choices made under the specified constraints as its primary goal is to define the non-functional requirements of a system and define the environment. It is often carried out in parallel with some specification

activities and It includes the high-level design of modular components, their relationships and organization, and provides a foundation that one can build on to solve a problem.

8.3 Why is architecture important?

Barry Boehm says:

> *If a project has not achieved a system architecture, including its rationale, the project should not proceed to full-scale system development. Specifying the architecture as a deliverable enables its use throughout the development and maintenance process.*

Why is architecture important and why is it worthwhile to invest in the development of a architecture? Fundamentally, there are three reasons:

1. **Mutual communication.** Software architecture represents a common high-level abstraction of the system that most, if not all, of the system's stakeholders can use as a basis for creating mutual understanding, forming consensus, and communicating with each other.

   Each stakeholder of a software system (customer, user, project manager, coder, tester, and so on) is concerned with different characteristics of the system that are affected by its architecture. Architecture provides a common language in which different concerns can be expressed, negotiated, and resolved at a level that is intellectually manageable, even for large, complex systems. Without such language, it is difficult to understand large systems sufficiently to make the early decisions that influence both quality and usefulness.

2. **Early design decisions.** Software architecture represents the embodiment of the earliest set of design decisions about a system, and these early bindings carry weight far out of proportion to their individual gravity with respect to the system's remaining development, its service in deployment, and its maintenance life. It is also the earliest point at which the system to be built can be analyzed.

   An implementation exhibits an architecture if it conforms to the structural design decisions described by the architecture. This means that the implementation must be divided into the prescribed components, the components must interact with each other in the prescribed fashion, and each component must fulfill its responsibility to the other components dictated by the architecture.

   Resource allocation decisions also constrain implementation. These decisions may be invisible to implementers working on individual components. The constraints permit a separation of concerns that allows management decisions that make best use of personnel and computational capacity. Component builders must be fluent in the specification of their individual components but not in architectural trade-offs. Conversely, the architects need not be experts in all

aspects of algorithm design or the intricacies of the programming language, but they are the ones responsible for architectural trade-offs.

Not only does architecture prescribe the structure of the system being developed, but it also engraves that structure on the structure of the development project. The normal method of dividing up the labor in a large system is to assign different portions of the system to different groups to construct. This is called the work breakdown structure of a system. Because the system architecture includes the highest level decomposition of the system, it is typically used as the basis for the work breakdown structure. Specifically, the module structure is most often the basis for work assignments. The work breakdown structure, in turn, dictates units of planning, scheduling, and budget, as well as inter-team communications channels, configuration control and file system organization, integration and test plans and procedures. Teams communicate with each other in terms of the interface specifications to the major components. The maintenance activity, when launched, will also reflect the software structure, with teams formed to maintain specific structural components.

A side effect of establishing the work breakdown structure is to effectively freeze the software architecture, at least at the level reflected in the work breakdown. A group that is responsible for one of the subsystems will resist having its responsibilities distributed across other groups. If these responsibilities have been formalized in a contractual relationship, changing responsibilities could become expensive. Tracking progress on a collection of tasks that is being distributed would also become much more difficult.

Thus, once the architecture's module structure has been agreed on, it becomes almost impossible for managerial and business reasons to modify it. This is one argument (among many) for carrying out extensive analysis before freezing the software architecture for a large system.

Is it possible to tell that the appropriate architectural decisions have been made (i.e., if the system will exhibit its required quality attributes) without waiting until the system is developed and deployed? If the answer were "no," choosing an architecture would be a hopeless task: random architecture selection would perform as well as any other method. Fortunately, it is possible to make quality predictions about a system based solely on an evaluation of its architecture.

3. **Reusable abstraction of a system.** Software architecture embodies a relatively small, intellectually graspable model for how the system is structured and how its components work together; this model is transferable across systems; in particular, it can be applied to other systems exhibiting similar requirements, and can promote large scale reuse.

   In an architecture-based development of a product line, the architecture is in fact the sum of the early design decisions. System architects choose an architecture

(or a family of closely related architectures) that will serve all envisioned members of the product line by making design decisions that apply across the family early and by making other decisions that apply only to individual members late. The architecture defines what is fixed for all members of the family and what is variable.

A family-wide design solution may not be optimal for all systems derived from it, but the quality gained and labor savings realized through architectural-level reuse may compensate for the loss of optimality in particular areas. On the other hand, reusing a family-wide design reduces the risk that a derived system might have an inappropriate architecture. Using a standard design reduces both risk and development costs, at the risk of non-optimality.

8.4 Architectural Attributes

Software architecture must address the non-functional as well as the functional requirements of the software system. This includes performance, security, safety, availability, and maintainability. Following are some of the architectural design guidelines that can help in addressing these challenges.

- Performance
  - Performance can be enhanced by localising operations to minimise sub-system communication. That is, try to have self-contained modules as much as possible so that inter-module communication is minimized.
- Security
  - Security can be improved by using a layered architecture with critical assets put in inner layers.
- Safety
  - Safety-critical components should be isolated
- Availability
  - Availability can be ensured by building redundancy in the system and having redundant components in the architecture.
- Maintainability
  - Maintainability is directly related with simplicity. Therefore, maintainability can be increased by using fine-grain, self-contained components.

8.5 Architectural design process

Just like any other design activity, design of software architecture is a creative and iterative process. This involves performing a number of activities, not necessarily in any particular order or sequence. These include system structuring, control modeling, and modular decomposition. These are elaborated in the following paragraphs.

- System structuring

System structuring is concerned with decomposing the system into interacting sub-systems. The system is decomposed into several principal sub-systems and communications between these sub-systems are identified. The architectural design is normally expressed as a block diagram presenting an overview of the system structure. More specific models showing how sub-systems share data, are distributed and interface with each other may also be developed. A sub-system is a system in its own right whose operation is independent of the services provided by other sub-systems. A module is a system component that provides services to other components but would not normally be considered as a separate system.

- Control modelling

  Control modelling establishes a model of the control relationships between the different parts of the system.

- Modular decomposition

  During this activity, the identified sub-systems are decomposed into modules.

This design process is further elaborated in the following section where architectural views are discussed.

Lecture No. 23

8.6 Architectural Views

Software architecture defines the high level structure of the software by putting together a number of architectural elements in an organized fashion. These elements are chosen to satisfy the functional as well as non-functional requirements of the system. Perry and Wolfe proposed the following formula for software architecture:

Software architecture = {Elements, Forms, Rationale}

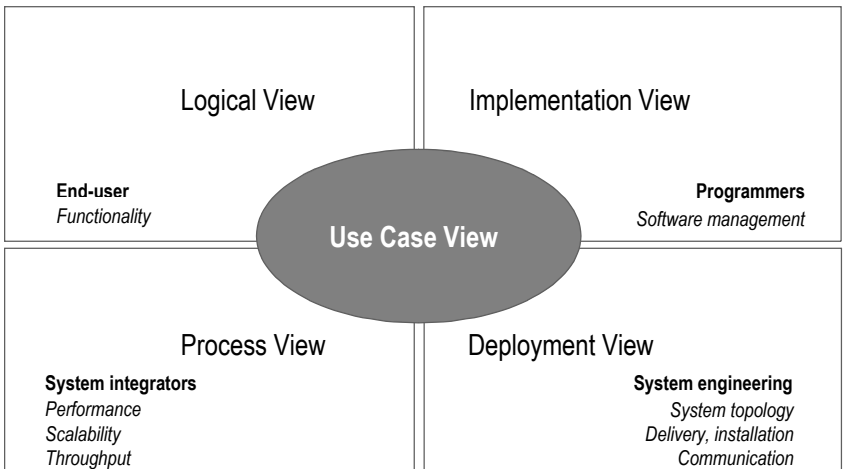
That is, a software architecture is a set of elements which have a certain form. These elements are further divided into three categories. These are: data elements, processing elements, and connecting elements. The data elements contain the information that is used and transformed in the system; the processing elements process the data elements and specify the transformation functions, and connecting elements connect different pieces of architecture together. These can themselves be data or processing elements.

This formula was modified by Boehm as follows:

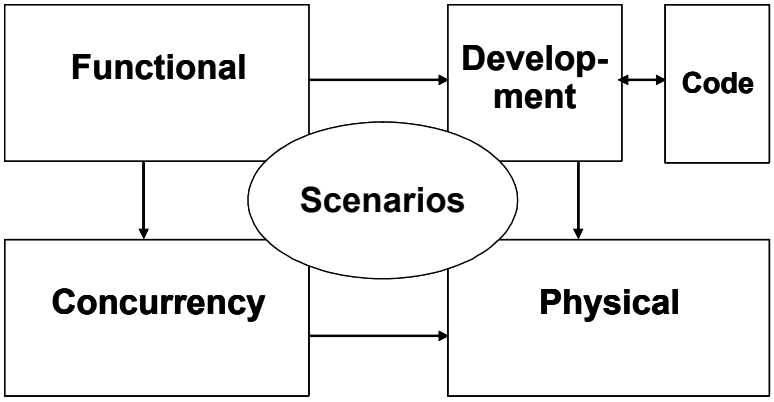
Software architecture = {Elements, Forms, Rationale/Constraints}

Krutchen proposed that the software architecture model should be composed of multiple views as a software architecture deals with abstraction, with decomposition and composition, with style and esthetics as well as requirements from different stake holders. His architectural model is known as Krutchen's 4+1 architectural view model. As evident, this model proposes the development of 5 main views namely the logical view, the process view, the physical view, the development view, and the use case view. The logical view is the object model of the design, the *process* view captures the concurrency and synchronization aspects of the design, the *physical* view documents the mapping(s) of the software onto the hardware and reflects its distributed aspect, the *development* view describes the static organization of the software in its development environment, and the use case view uses selected use cases or scenarios to validate that the architecture supports the required functionality.

This model is shown in the following diagram.



This model has been slightly modified by Clements et. al. and is shown in the following diagram.



Clement's modified version of Krutchen's 4+1 Architectural View Model

In this model, the architecture is again prepared and analyzed from 5 different perspectives. The 4 main views are Functional View, the Concurrency View, the Physical View, and the Development View. Code view is not present in the original Krutchen model and is basically an extension of the development view.

The Functional View comprises of various different functions provided by the system, key system abstraction, and the domain elements. It connects different dependencies and data flows into a single view. This view can be used by the domain engineers, product-line engineers, as well as the end users of the system. It can be used for understanding the functionality provided by the system, modifiability of the system, reusability, tool support, and allocation of work.

The Development View documents the different files and directories in the system and users of this view include the development and the configuration management staff as well as the project managers. The major uses of this view include maintenance, testing, configuration management, and version control.

The components of the Code View include classes, objects, procedures, functions, subsystems, layers, and modules. It documents the calling as well as the containing hierarchy of different components of the system. Its primary users are the programmers and designers of the system. The primary intent of developing this view is to use it for maintenance and portability.

Concurrency View intends to document different parallel processes and threads in the system. Its main focus is on event synchronization and parallel data flows. It is used primarily by integrators, performance engineers, and testers. The main purpose of building this view is to identify ways and means to improve performance by highlighting possible opportunities for parallelism.

The Physical View depicts the physical organization of this system and how this system will be physically deployed. This includes different processors, sensors, and storage devices used by the system. This view connects various network elements and communication devices. The primary users of this system include hardware and system engineers. The view is developed with an intention to document and analyze system delivery and installation mechanism. The view is also used in understanding and analyzing issues pertaining to system performance, availability, scalability, and security.

Finally, there are Scenarios. Scenarios are basically use cases which describe the sequences of responsibilities and change cases which are changes to the system. As stated earlier, the first objective of developing a system is to fulfill the functional requirements setout for the system. These functional requirements are written in the form of use cases. Therefore, scenarios are used to understand and validate the system. Their secondary purpose is to communicate the design to the other users of the system. Scenarios tie all the view together into an integrated system. They are also used to understand the dynamic behavior of the system and understand the limits of design.

This is summarized in the following table:

View	Components	Users	Rationale
Functional View	functions, key system abstractions, domain elements	domain engineers, product-line designers, end users	functionality, modifiability, product lines/reusability, tool support, work allocation
Code View	classes, objects, procedures, functions, subsystems, layers, modules	programmers, designers, reusers	modifiability/maintainability, portability, subsetability
Development View	files, directories	managers, programmers, configuration managers	managers, programmers, configuration managers
Physical View	CPUs, sensors, storage	hardware engineers, system engineers	system delivery and installation, performance, availability, scalability, security
Concurrency View	processes, threads	performance engineers, integrators, testers	performance, availability

What Are Views Used For?

Views are an engineering tool to help achieve desired system qualities. Each view provides an engineering handle on certain quality attributes. In some systems, distinct views collapse into one (e.g., the concurrency and physical views may be the same for small systems.). Views are also used as documentation vehicle for current development and future development. Users of views include both managers and customers. For these reasons views must be *annotated* to support analysis. This annotation can be achieved with the help of scenarios and design *rationale*.

Structures can also be used to document how the current system was developed and how future development should occur. This information is needed by managers and customers.

Although one often thinks about a system's structure in terms of functionality, there are other system properties in addition to functionality (such as physical distribution, process communication, and synchronization) that must be reasoned about at an architectural level. Each structure provides a method for reasoning about some of the relevant quality attributes. The uses structure, for instance, must be engineered (not merely recorded) to build a system that can easily be extended or contracted. The calls structure is engineered to reduce bottlenecks. The module structure is engineered to produce modifiable systems, and so on. Each structure provides a different view into the system and a different leverage point for design to achieve desired qualities in the system.

Hierarchical Views

Every view is potentially hierarchical, e.g. *functional view could contain* sub-functions. Similarly *development view* contains directories which contain files. C*ode view* would have modules and systems that contain sub-modules and sub-systems respectively. C*oncurrency view* contains processes that are further subdivided into threads. Finally, *physical view* clusters contain computers which contain processors.

Architectural views are related to each other in complicated ways. One has to choose the views that are useful to the system being built and to the achievement of qualities that are important for that particular application. The architectural views should be hierarchical (where needed) and should contain enough annotated information to support desired analyses.

Architectural styles

The architectural model of a system may conform to a generic architectural model or style. An awareness of these styles can simplify the problem of defining system architectures. However, most large systems are heterogeneous and do not follow a single architectural style.

Each style describes a system category that encompasses:

(1) a set of components (e.g., a database, computational modules) that perform a function required by a system,
(2) a set of connectors that enable "communication, coordination and cooperation" among components,
(3) constraints that define how components can be integrated to form the system, and

semantic models that enable a designer to understand the overall properties of a system by analyzing the known properties of its constituent parts.

Lecture No. 24

8.7 Architectural models

Like analysis models, many different kinds of architectural models are developed during the architectural design process. Static structural model shows the major system components while a dynamic process model shows the process structure of the system. Interface models are developed to define sub-system interfaces.

8.8 Architectural Styles

Architectural design may be based upon a certain pattern of model. These different patterns are also called architectural styles. These styles have different characteristics and attributes and can be useful to solve problems related to a particular situation of requirement. Among the many styles, the most commonly practiced are the following:

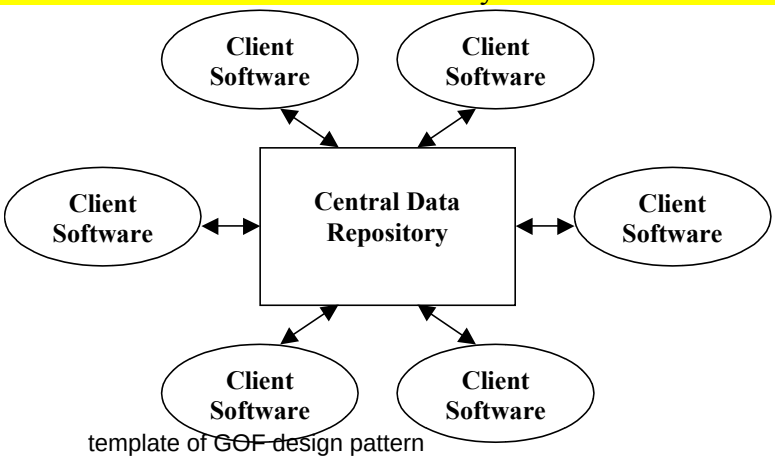
- Data-centered architectures
- Client Server Architecture and its variations
- Layered architectures
- Reference Architecture

Data-Centered or the repository model

In any system, sub-systems need to exchange information and data. This may be done in two ways:

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

When large amounts of data are to be shared, the repository model of sharing is most commonly used. This model has been extensively used in the main-frame based



template of GOF design pattern includes

application. The model is depicted in the following diagram:

Repository model characteristics

- Advantages

  Repository model is an efficient way to share large amounts of data. In this case sub-systems need not be concerned with how data is produced Centralised management e.g. backup, security, etc. This model also provides a global view of the system and the sharing model is published as the repository schema.

- Disadvantages

  Repository model suffers from a number of disadvantages. First of all, sub-systems must agree on a repository data model. This inevitably leads to a compromise. The major problem however is that data evolution is difficult and expensive. There is also little scope for implementing specific management policies. It is also difficult to distribute efficiently

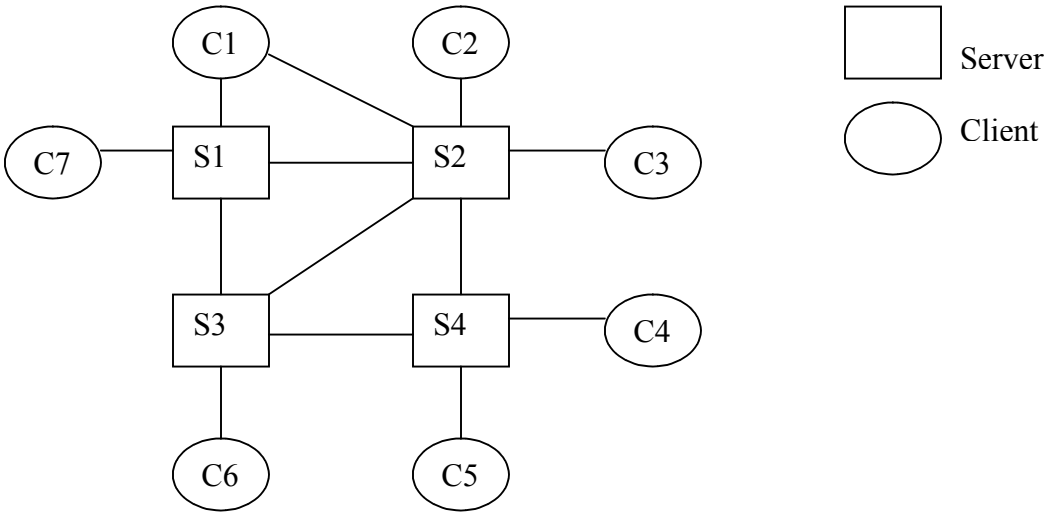
8.9 Client-server model

Client server model tries to distribute data and processing. This is a shift from main-frame based applications where both the data management and the processing of data used to be typically carried out by the same main-frame computer. In those applications, the user interface used to be a provided through a "dumb" terminal which did not have much processing power. With the availability of the cheaper but power machines, it was possible to shift some load from the back-end computer to other smaller machines.

The client-server model is a distributed system model which shows how data and processing is distributed across a range of components. In this model, the application is modeled as a set of services that are provided by servers and a set of clients that use these services. The system is organized as a set of stand-alone servers which provide specific services such as printing, data management, etc. and a set of clients which call on these

services. These clients and servers are connected through a network which allows clients to access servers. Clients and servers are logical processes (not always physical machines). Clients know the servers but the servers do not need to know all the clients and the mapping of processes to processors is not always 1:1.

The following diagram depicts a general client-server organization.



8.10Client/Server Software Components

A typical client-server architecture based system is composed of a number of different components. These include user interaction/presentation subsystem, application subsystem, database management subsystem, and middleware. The application subsystem implements requirements defined by the application within the context of the operating environment. In this case the application components may reside on either client or server side. Middleware provides the mechanism and protocols to connect clients with the servers.

Representative Client/Server Systems

Following are some of the representative server types in a client-server systems.

- File servers

  In this case, client requests selected records from a file and the server transmits records to client over the network.
- Database servers

  In this case, client sends requests, such as SQL queries, to the database server, the server processes the request and returns the results to the client over the network.
- Transaction servers

  In this configuration, client sends requests that invokes remote procedures on the server side, server executes procedures invoked and returns the results to the client.
- Groupware servers

  Groupware servers provide set of applications that enable communication among clients using text, images, bulletin boards, video, etc.

Client-server characteristics

- Advantages

  The main advantage of the client-server architecture is that it makes effective use of networked systems. Instead of building the system with very expensive large computers and hardware devices, cheaper hardware may be used to gain performance and scalability. In addition, adding new servers or upgrading existing servers becomes easier. Finally, distribution of data is straightforward in this case.

- Disadvantages

  The main disadvantage of this model is that there is no standard way of sharing data, so sub-systems may use different data organisation. Hence data interchange may be inefficient. From a management point of view, each server needs attention and hence there is redundant management in each server. Finally there is no central register of names and services - it may be hard to find out what servers and services are available.

8.11 Representative Client/Server Configurations

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

Thin Client Model

This model was initially used to migrate legacy systems to client server architectures. In this case the legacy system may act as a server in its own right and the GUI may be implemented on a client. It chief disadvantage is that it places a heavy processing load on both the server and the network.

Fat Client Model

With advent of cheaper and more powerful hardware, people thought of using the processing power of client side machines. So the fat client model came into being. In this model, more processing is delegated to the client as the application processing is locally extended. It is suitable for new client/server systems when the client system capabilities are known in advance. It however is more complex than thin client model with respect to management issues. Since the client machine now also has a significant part of the application resident on it, new versions of each application need to be installed on every client.

Lecture No. 25

Zero Install

As discussed earlier, fat-client architecture posed major challenges in terms of installation and maintenance of the client side of the application, especially when there are large number of client machines. So the idea behind zero install architecture is to develop a system where no installation on the client side is needed. This can only be done when there is no or little processing done at the client side. This is basically a trade-off between using the computing power available at the client machine versus maintenance overhead. This in essence takes us back to an architecture which is similar to thin-client architecture. There is little difference though. In the classical thin-client architecture, the entire processing is carried-out by a single server, in the case of zero-install, the network environment is used to distribute server side processing by adding a number of servers which share processing load. Web-based application where the web-pages are put on a web-server is an example of this type of architecture. In this case, whenever there is a change, the web page is updated accordingly. Now when the user logs in, he gets to use the modified version of the application without any changes or updates at the client side.

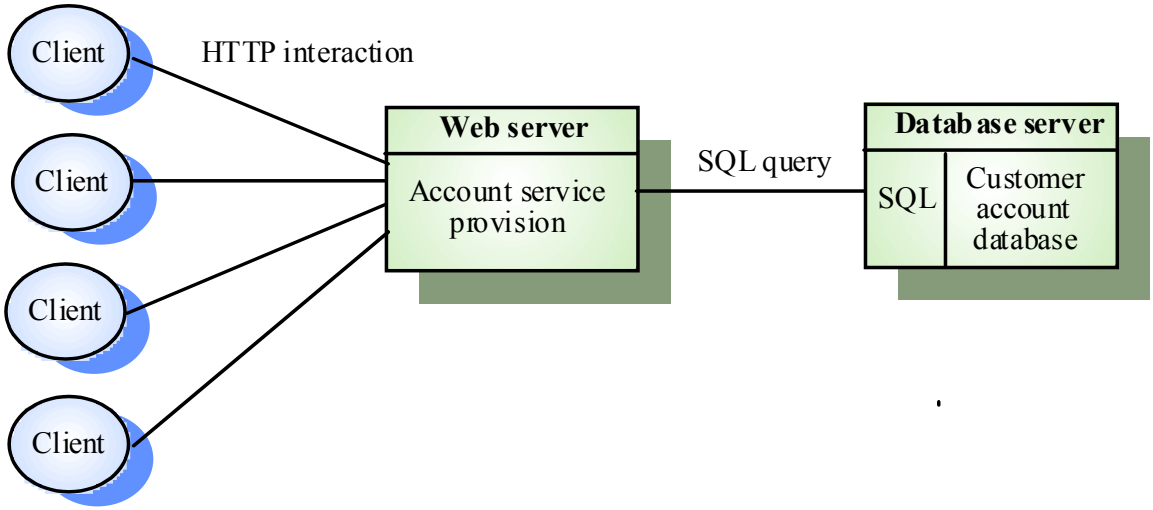
N-Tier architecture

N-tier architecture stems from the struggle to find a middle ground between the fat-client architecture and the thin-client architecture. In this case the idea is to enhance scalability and performance by distributing both the data and the application using multiple server machines. This could involve different types of servers such as application server, web server, and DB server. Three-tier architecture which is explained below is a specialized form of this architecture.

Three-tier Architecture

In this architecture, each application architecture layers (presentation, application, database) may run on separate processors. It therefore allows for better performance than a thin-client approach. It is simpler to manage than fat client approach and highly scalable (as demands increase add more servers).

A typical 3-tier architecture is depicted in the following diagram.

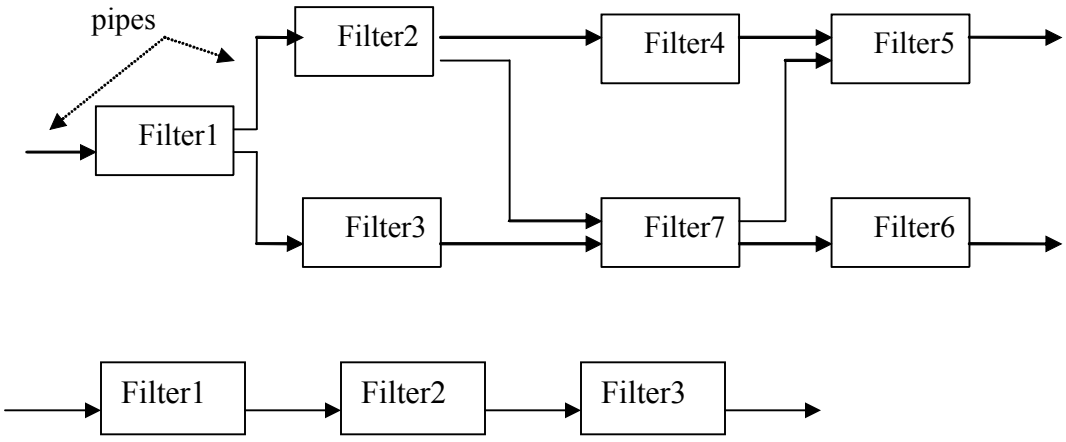


N-tier architecture generalizes the concepts of 3-tier architecture. In this case the system architecture may have more than 3 layers. That is, in n-tier architecture, in order to increase performance, we may distribute the application over different servers by putting different subsystems on different servers.

8.12 Data Flow or Pipes and Filters Architecture

This architecture is very similar to data flow diagrams. This is used when the input data is processed through a series of transformations to yield the desired output. It is also known as pipes and filters architecture where each processing step is called a filter and the connecting link from one process to the other is called the pipe through which the information flows from one process to the other. An important aspect of this model is that each filter works independently of others and does not require knowledge of the working of any of the other filters including its neighbours.

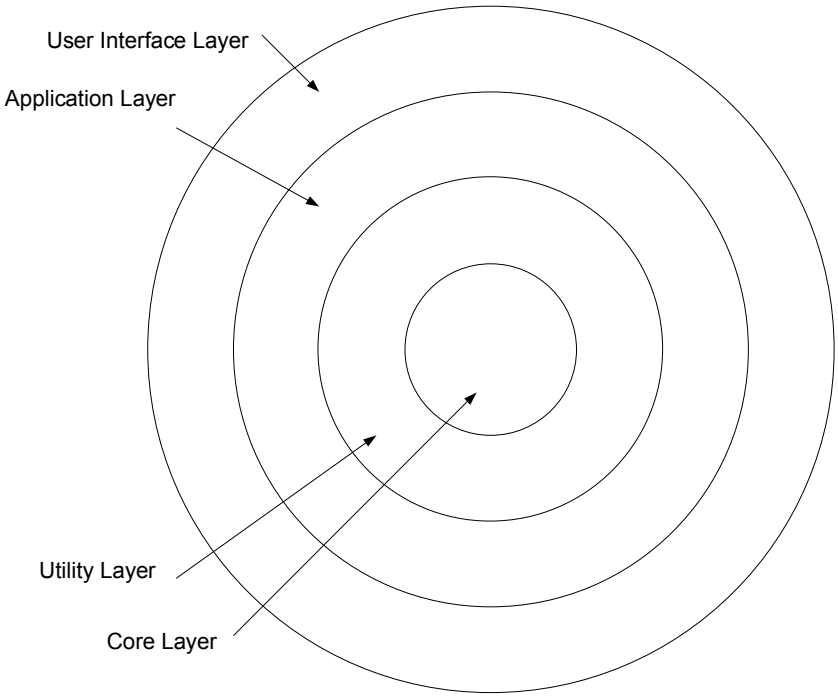
If the dataflow has only a single sequence of processes with no alternative or parallel paths, then it is called batch sequential. These models are depicted in the following diagrams.



Batch Sequential

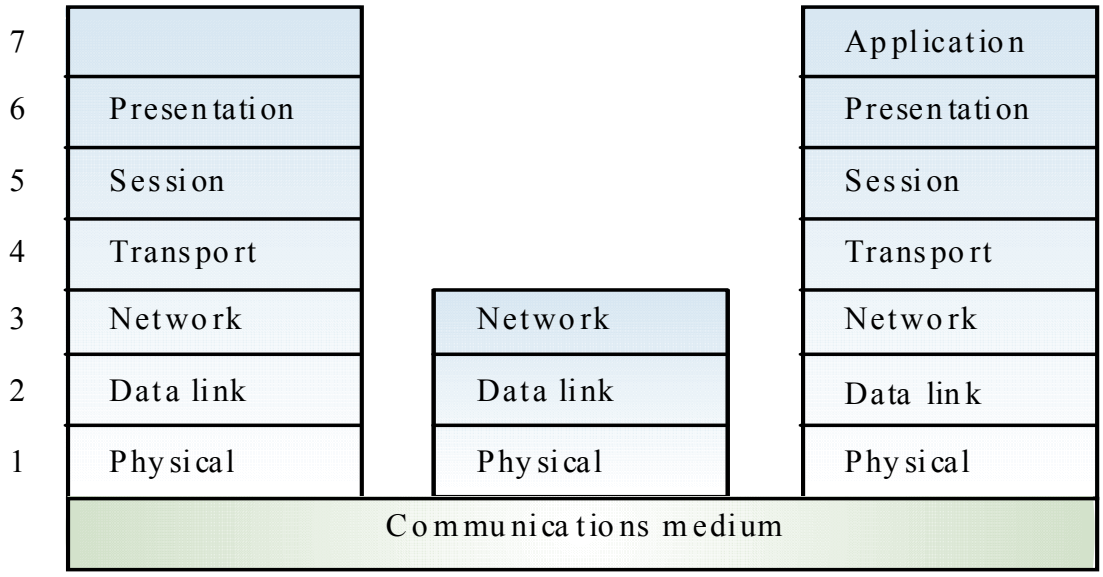
Layered Architecture

As the name suggests, a layered architecture has many different layers. One typical example of a layered architecture is an operating system where different layers are used to provide services and functionality and the inner layers are closer to the machine hardware than the outer layers. In this way, each layer isolates the outer layer from inner complexities. In order to work properly, the outer layer only needs to know the interface provided by the inner layer. If there are any changes in the inner layer, as long as the interface does not change, the outer layer is not affected. This scheme tremendously portability of the system. The basic layered architecture is depicted in the following diagram.



8.13 Reference architectures

Reference architecture is not a physical architecture. It is only a reference for defining protocols and designing and implementing systems developed by different parties. Reference models are derived from a study of the application domain rather than from existing systems. It may be used as a basis for system implementation or to compare different systems. It acts as a standard against which systems can be evaluated. One very common example of such a reference model is the OSI model which is a layered model for communication systems. The success of this kind model is evident from the success of the Internet where all kinds of heterogeneous systems can talk to each other only because all of them use the same reference architecture.



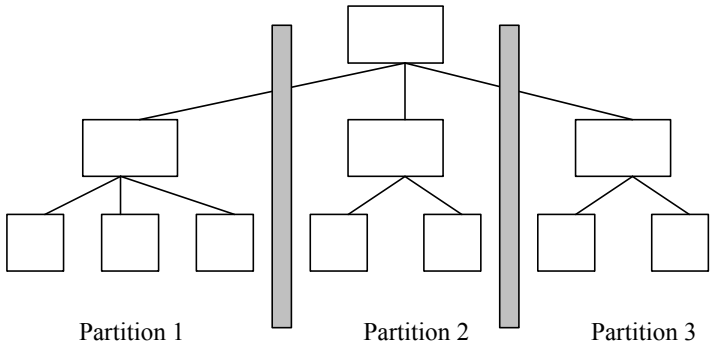
OSI reference model

8.14 Partitioning the Architecture

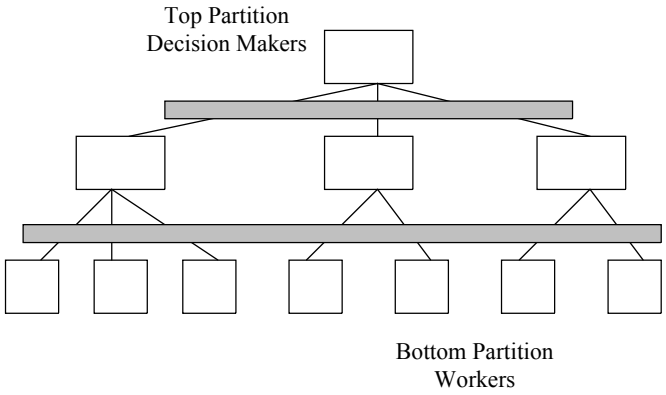
Partitioning of architecture is an important concept. What we basically want to do is distribute the responsibilities to different subsystems so that we get a software system which is easy to maintain. Partitioning results in a system that suffers from fewer side effects. This ultimately means that we get a system that is easier to test and extend and hence is easier to maintain.

Partitioning of an architecture may be "horizontal" and/or "vertical".

In the horizontal partitioning we define separate branches of the module hierarchy for each major function and control modules are used to coordinate communication between functions. This concept is depicted in the following diagram.



Vertical partitioning divides the application from a decision making perspective. The architecture is partitioned in horizontal layers so that decision making and work are stratified with the decision making modules residing at the top of the hierarchy and worker coming at the bottom. This partitioning is also known as factoring and the general model is depicted in the following diagram.



8.15 Analyzing Architecture design

In a given system, the required characteristics may conflict. Trade-offs seek optimal combinations of properties based on cost/benefit analysis. So the analysis requires an understanding of what is required and the relative priority of the attributes has to be established. The following sequence of steps provides a guideline for performing architectural analysis.

1. Collect scenarios.
2. Elicit requirements, constraints, and environment description.
3. Describe the architectural styles/patterns that have been chosen to address the scenarios and requirements. These views include module view, process view, and data flow view.
4. Evaluate quality attributes by considered each attribute in isolation.
5. Identify the sensitivity of quality attributes to various architectural attributes for a specific architectural style.

Critique candidate architectures (developed in step 3) using the sensitivity analysis conducted in step 5.

Lecture No. 26

Introduction to Design Patterns

Design Patterns

Christopher Alexander says, "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice." A Pattern Language: Towns/Buildings/Construction, 1977

Even though Alexander was talking about patterns in buildings and towns, what he says is true about object-oriented design patterns. Our solutions are expressed in terms of objects and interfaces instead of walls and doors, but the core of both kinds of patterns is a solution to a problem in a context.

Design Patterns defined

"Description of communicating objects and classes that are customized to solve a general design in a particular context."

Patterns are devices that allow programs to share knowledge about their design. In our daily programming, we encounter many problems that have occurred, and will occur again. The question we must ask our self is how we are going to solve it *this* time. Documenting patterns is one way that you can reuse and possibly share the information that you have learned about how it is best to solve a specific program design problem.

Essay writing is usually done in a fairly well defined form, and so is documenting design patterns. The general form for documenting patterns is to define items such as:

- The motivation or context that this pattern applies to.
- Prerequisites that should be satisfied before deciding to use a pattern.
- A description of the program structure that the pattern will define.
- A list of the participants needed to complete a pattern.
- Consequences of using the pattern...both positive and negative.
- Examples!

Historical perspective of design patterns

The origin of design patterns lies in work done by an architect named *Christopher Alexander* during the late 1970s. He began by writing two books, *A Pattern*

Language[Alex77] and *A Timeless Way of Building* [Alex79] which, in addition to giving examples, described his rationalle for documenting patterns.

The pattern movement became very quiet until 1987 when patterns appeared again at an OOPSLA conference. Since then, many papers and presentations have appeared, authored by people such as Grady Booch, Richard Helm, and Erich Gamma, and Kent Beck. From then until 1995, many periodicals, featured articles directly or indirectly relating to patterns. In 1995, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides published Design Patterns: Elements of Reusable Object-Oriented Software [Gamma95], which has been followed by more articles in trade journals.

The concept of design patterns is not new as we can find a number of similar pursuits in the history of program designing and writing. For instance, Standard Template Library (STL) is a library of reusable components provided by C++ compilers. Likewise, we use algorithms in data structures that implement typical operations of manipulating data in data structures. Another, similar effort was from Peter Coad whose patterns are known for object-oriented analysis and design.
Anti-patterns is another concept that corresponds to common mistakes in analysis and design. These are identified in order to prevent potential design and analysis defects from entering into the design. Another, similar concept is object-oriented framework that is a set of cooperative classes that make up reusable design of a system. Framework dictates the architecture of the software and describes the limitations and boundaries of architecture.
With this introduction, we now describe the format that has been adopted in GoF book for describing various design patterns.

GOF Design Pattern Format

The basic template includes ten things as described below

Name

- Works as idiom
- Name has to be meaningful

Problem

- A statement of the problem which describes its intent
- The goals and objectives it wants to reach within the given context

Context

- Preconditions under which the problem and its solutions seem to occur
- Result or consequence
- State or configuration after the pattern has been applied

Forces

- Relevant forces and constraints and their interactions and conflicts.
- motivational scenario for the pattern.

Solution

- Static and dynamic relationships describing how to realize the pattern.
- Instructions on how to construct the work products.

- 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 GoF format.

Lecture No. 27

Observer Pattern

Name

- Observer

Basic intent

- It is intended to define a many to many relationship between objects so that when one object changes state all its dependants are notified and updated automatically.
- Dependence/publish-subscribe mechanism in programming language
    - Smalltalk being the first pure Object Oriented language in which observer pattern was used in implementing its Model View Controller (MVC) pattern. It was a publish-subscribe mechanism in which views (GUIs) were linked with their models (containers of information) through controller objects. Therefore, whenever underlying data changes in the model objects, the controller would notify the view objects to refresh themselves and vice versa.
    - MVC pattern was based on the observer pattern.

Motivation

- It provides a common side effect of partitioning a system into a collection of cooperating classes that are in the need to maintain consistency between related objects.

Description

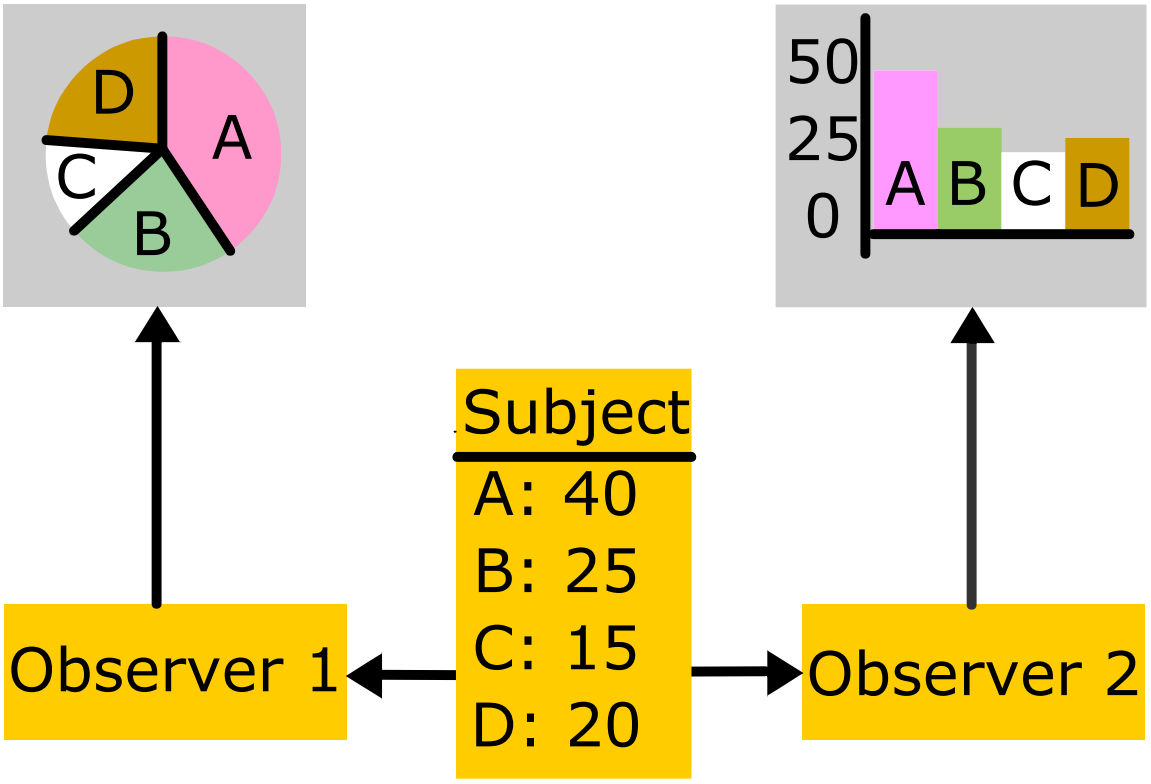
- This can be used when multiple displays of state are needed.

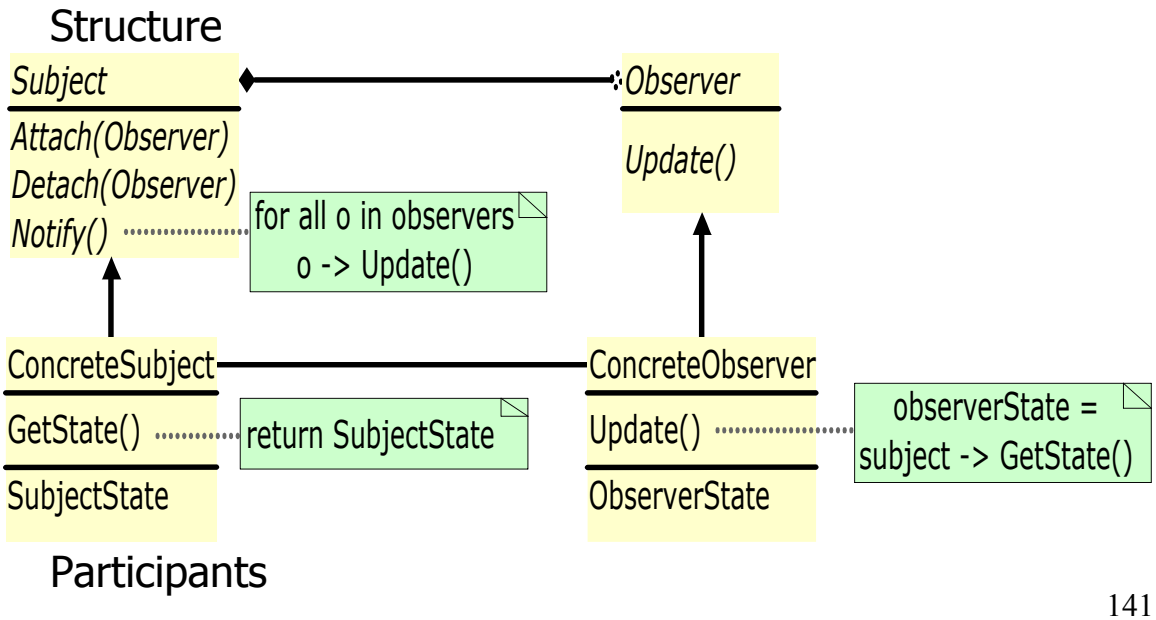
Consequences

- **Optimizations to enhance display performance are impractical.**

Example implementation of Observer PatternObject Model

Many graphical user interface toolkits separate the presentational aspects of the user interface from the underlying application data. Classes defining application data and presentations can be reused independently. They can work together, too. Both a spreadsheet object and bar chart object can depict information in the same application data object using different presentations. The spreadsheet and the bar chart don't know about each other, thereby letting you reuse only the one you need. But they behave as though they do. When the user changes the information in the spreadsheet, the bar chart reflects the changes immediately, and vice versa.





Subject

- Knows its observers. Any number of Observer objects may observe a subject.
- Provides an interface for attaching and detaching Observer objects.

Observer

- Defines an updating interface for objects that should be notified of changes in a subject.

ConcreteSubject

- Stores state of interest to concreteObserver objects.
- Sends a notification to its observers when its state changes.

ConcreteObserver

- Maintains a reference to a ConcreteSubject object.
- Stores state state that should stay consistent with the subject's.
- Implements the Observer updating interface to keep its state consistent with the subject's.

Singleton Pattern

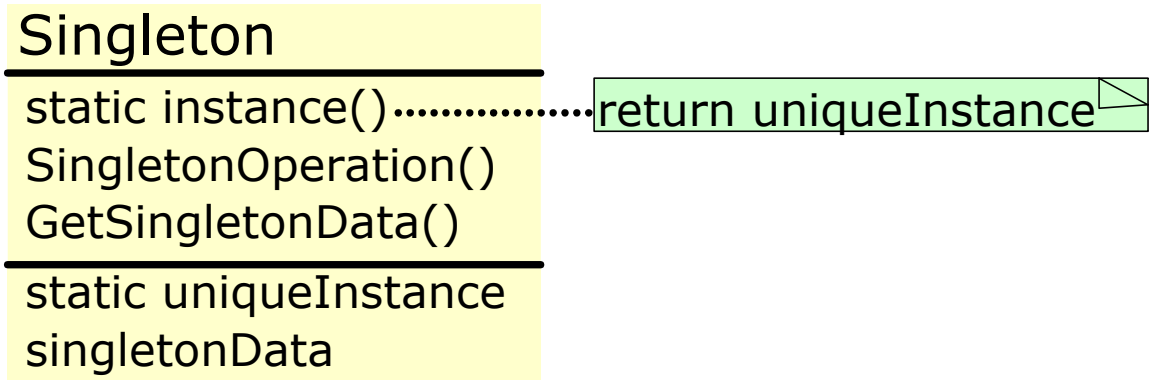
Intent

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

Applicability

- Singleton pattern should be used when there must be exactly one instance of a class and it must be accessible to clients from a well-known access point.
- Singleton pattern should be used when controlling the total number of instances that would be created for a particular class.
- Singleton pattern should be used when the sole instance should be extensible by sub classing and clients should be able to use an extended instance without modifying their code.

Structure



Participants

Singleton

- Defines an instance operation that lets clients access its unique instance. Instance is a class operation (that is, a class method in Smalltalk and a static member function in C++).
- May be responsible for creating its own unique instance.

Singleton Pattern Example

The Singleton class is declared as

```
class Singleton {
public:
    static Singleton* Instance();
protected:
    Singleton();
private:
    static Singleton* _instance;
};
```

The corresponding implementation is

```
Singleton* Singleton::_instance = 0;

Singleton* Singleton::Instance(){
     if (_instance == 0) {
        _instance = new Singleton;
     }
     return _instance;
}
```

Clients access the singleton exclusively through the Instance member function. The variable _instance is initialized to 0, and the static member function Instance returns its value, initializing it with the unique instance if it is 0.

Façade Pattern

Intent

- It provides a unified interface to a set of interfaces in a sub-system.
- Façade defines a higher level interface that makes a subsystem easier to use

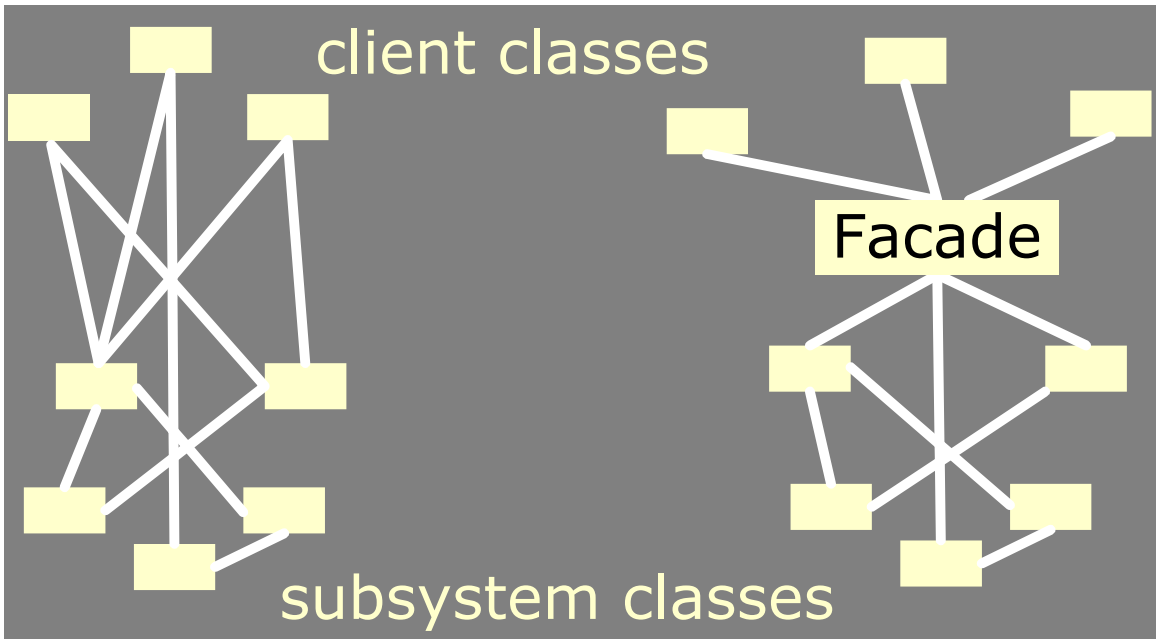
Applicability

- You would use façade when you want to provide a simple interface to a complex sub-system.
- You would use façade pattern when there are many dependencies between clients and the implementation classes of an abstraction.
- You should introduce a façade to decouple the system from clients and other subsystems.
- You want to layer your subsystem.
- You would use façade when you want to provide a simple interface to a complex sub-system

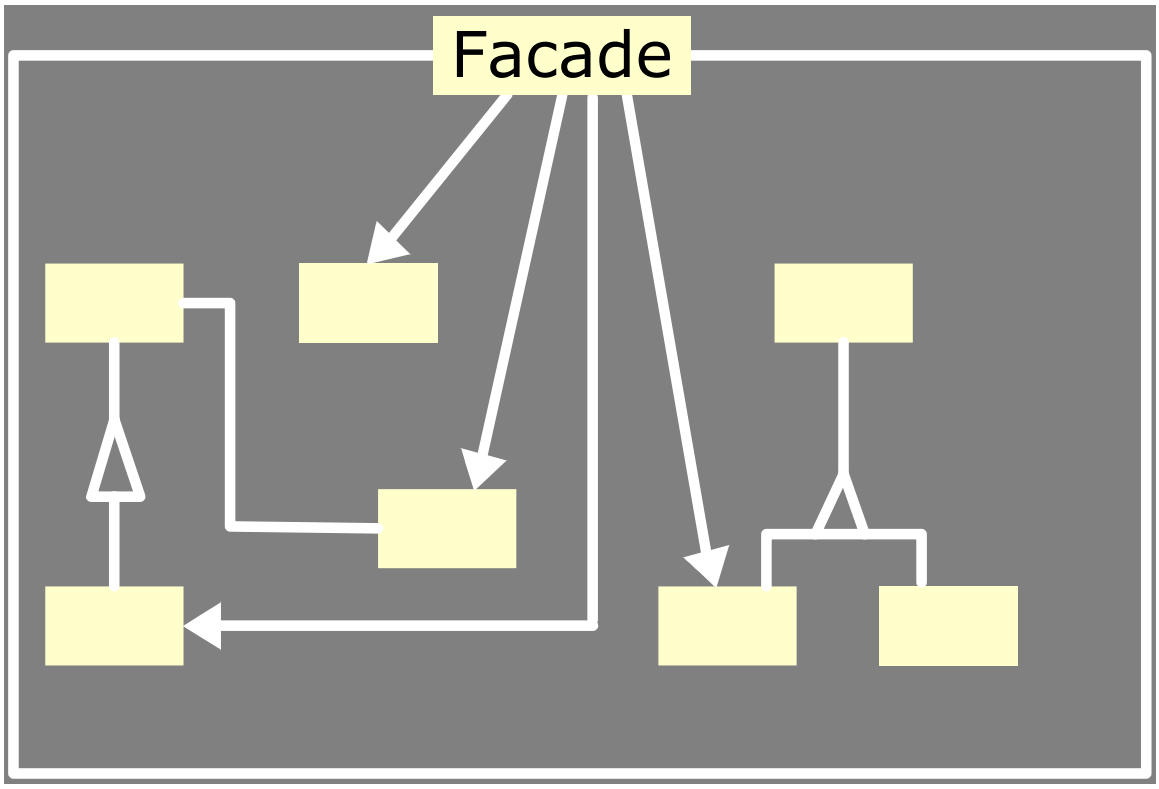
- You would use façade pattern when there are many dependencies between clients and the implementation classes of an abstraction
- You should introduce a façade to decouple the system from clients and other subsystems
- You want to layer the subsystem

Abstract example of façade

Structuring a system into subsystems helps reduce complexity. A common design goal is to minimize the communication and dependencies between subsystems. One way to achieve this goal is to introduce a façade object that provides a single, simplified interface to the more general facilities of a subsystem.



Structure



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

10.1 Maintainable Code

As we have discussed earlier, in most cases, maintainability is the most desirable quality of a software artifact. Code is no exception. Good software ought to have code that is easy to maintain. Fowler says, "*Any fool can write code that computers can understand, good programmers write code that humans can understand."* That is, it is not important to write code that works, it is important to write code that works and is easy to understand so that it can be maintained. The three basic principles that guide maintainability are: simplicity, clarity, and generality or flexibility. The software will be easy to maintain if it is easy to understand and easy to enhance. Simplicity and clarity help in making the code easier to understand while flexibility facilitates easy enhancement to the software.

Self Documenting Code

From a maintenance perspective, what we need is what is called ***self documenting code***. A *self documenting code* is a code that explains itself without the need of comments and extraneous documentation, like flowcharts, UML diagrams, process-flow state diagrams, etc. That is, the meaning of the code should be evident just by reading the code without having to refer to information present outside this code.

The question is: how can we write code that is self-documenting?

There are a number of attributes that contributes towards making the program self documented. These include, the size of each function, choice of variable and other identifier names, style of writing expressions, structure of programming statements, comments, modularity, and issues relating to performance and portability.

The following discussion tries to elaborate on these points.

Function Size

The size of individual functions plays a significant role in making the program easy or difficult to understand. In general, as the function becomes longer in size, it becomes more difficult to understand. Ideally speaking, a function should not be larger than 20 lines of code and in any case should not exceed one page in length. From where did I get this number 20 and one page? The number 20 is approximately the lines of code that fit on a computer screen and one page of course refers to one printed page. The idea behind these heuristics is that when one is reading a function, one should not need to go back and forth from one screen to the other or from one page to the other and the entire context should be present on one page or on one screen.

Identifier Names

Identifier names also play a significant role in enhancing the readability of a program. The names should be chosen in order to make them meaningful to the reader. In order to understand the concept, let us look at the following statement.

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

In this particular case, the meanings of the condition in the if-statement are not clear and we had to write a comment to explain it. This can be improved if, instead of using x, we use a more meaningful name. Our new code becomes:

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

The situation has improved a little bit but the semantics of the condition are still not very clear as the meaning of 0 is not very clear. Now consider the following statement:

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

We have improved the quality of the code by replacing the number 0 with a named constant NEW_NUMBER. Now, the semantics are clear and do not need any extra comments, hence this piece of code is self-documenting.

10.2 Coding Style Guide

Consistency plays a very important role in making it self-documenting. A consistently written code is easier to understand and follow. A coding style guide is aimed at improving the coding process and to implement the concept of standardized and relatively uniform code throughout the application or project. As a number of programmers participate in developing a large piece of code, it is important that a consistent style is adopted and used by all. Therefore, each organization should develop a style guide to be adopted by its entire team.

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

10.3 Naming Conventions

Hungarian Notation was first discussed by Charles Simonyi of Microsoft. It is a variable naming convention that includes information about the variable in its name (such as data type, whether it is a reference variable or a constant variable, etc). Every company and programmer seems to have their own flavor of Hungarian Notation. The advantage of Hungarian notation is that by just looking at the variable name, one gets all the information needed about that variable.

Bicapitalization or *camel case* (frequently written *CamelCase*) is the practice of writing compound words or phrases where the terms are joined without spaces, and every term is capitalized. The name comes from a supposed resemblance between the bumpy outline of the compound word and the humps of a camel. CamelCase is now the official convention for file names and identifiers in the Java Programming Language.

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

General Naming Conventions

General naming conventions for Java and C++

1. Names representing types must be nouns and written in mixed case starting with upper case.

```
Line, FilePrefix
```

2. Variable names must be in mixed case starting with lower case.

```
line, filePrefix
```

This makes variables easy to distinguish from types, and effectively resolves potential naming collision as in the declaration `Line line;`

3. Names representing constants must be all uppercase using underscore to separate words.

```
MAX_ITERATIONS, COLOR_RED
```

In general, the use of such constants should be minimized. In many cases implementing the value as a method is a better choice. This form is both easier to read, and it ensures a uniform interface towards class values.

```
int    getMaxIterations()//    NOT:    MAX_ITERATIONS    =    25
{
                              return                          25;
}
```

4. Names representing methods and functions should be verbs and written in mixed case starting with lower case.

```
getName(), computeTotalWidth()
```

5. Names representing template types in C++ should be a single uppercase letter.

```
template<class                        T>                      ...
template<class C, class D> ...
```

6. Global variables in C++ should always be referred to by using the :: operator.

::mainWindow.open() , ::applicationContext.getName()

7. Private class variables should have _ suffix.

```
class                                                    SomeClass
{
              private              int                  length_;
                                                             ...
}
```

Apart from its name and its type, the *scope* of a variable is its most important feature. Indicating class scope by using _ makes it easy to distinguish class variables from local scratch variables.

8. Abbreviations and acronyms should not be uppercase when used as name.

```
exportHtmlSource();          //    NOT:    xportHTMLSource();
openDvdPlayer();       // NOT: openDVDPlayer();
```

Using all uppercase for the base name will give conflicts with the naming conventions given above. A variable of this type would have to be named dVD, hTML etc. which obviously is not very readable.

9. Generic variables should have the same name as their type.

```
void setTopic (Topic topic)        // NOT: void setTopic (Topic value)
                                   // NOT: void setTopic (Topic aTopic)
                                       // NOT: void setTopic (Topic x)

void connect (Database database)  // NOT: void connect (Database db)
                                  // NOT: void connect (Database oracleDB)
```

Non-generic variables have a *role*. These variables can often be named by combining role and type:

```
Point              startingPoint,              centerPoint;
Name  loginName;
```

10. All names should be written in English.

```
fileName;    // NOT:   filNavn
```

11. Variables with a large scope should have long names, variables with a small scope can have short names. Scratch variables used for temporary storage or indices are best kept short. A programmer reading such variables should be able to assume that its value is not used outside a few lines of code. Common scratch variables for integers are *i*, *j*, *k*, *m*, *n* and for characters *c* and *d*.

12. The name of the object is implicit, and should be avoided in a method name.

```
line.getLength();    // NOT:  line.getLineLength();
```

The latter seems natural in the class declaration, but proves superfluous in use.

Specific Naming Conventions for Java and C++

1. The terms *get/set* must be used where an attribute is accessed directly.

```
employee.getName();
matrix.getElement (2, 4);
employee.setName (name);
matrix.setElement (2, 4, value);
```

2. is prefix should be used for boolean variables and methods.

```
isSet, isVisible, isFinished, isFound, isOpen
```

Using the *is* prefix solves a common problem of choosing bad boolean names like *status* or *flag*. *isStatus* or *isFlag* simply doesn't fit, and the programmer is forced to chose more meaningful names.

There are a few alternatives to the *is* prefix that fits better in some situations. These are *has*, *can* and *should* prefixes:

```
boolean hasLicense();
boolean canEvaluate();
boolean shouldAbort = false;
```

3. The term *compute* can be used in methods where something is computed.

```
valueSet.computeAverage();  matrix.computeInverse()
```

Using this term will give the reader the immediate clue that this is a potential time consuming operation, and if used repeatedly, he might consider caching the result.

4. The term *find* can be used in methods where something is looked up.

```
vertex.findNearestVertex();   matrix.findMinElement();
```

This gives the reader the immediate clue that this is a simple look up method with a minimum of computations involved.

5. The term *initialize* can be used where an object or a concept is established.

```
printer.initializeFontSet();
```

6. List suffix can be used on names representing a list of objects.

`vertex` (one vertex), `vertexList` (a list of vertices)

Simply using the plural form of the base class name for a list (`matrixElement` (one matrix element), `matrixElements` (list of matrix elements)) should be avoided since the two only differ in a single character and are thereby difficult to distinguish.
A *list* in this context is the compound data type that can be traversed backwards, forwards, etc. (typically a `Vector`). A plain array is simpler. The suffix *Array* can be used to denote an array of objects.

7. n prefix should be used for variables representing a number of objects.

```
nPoints, nLines
```

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

8. No suffix should be used for variables representing an entity number.

```
tableNo, employeeNo
```

The notation is taken from mathematics where it is an established convention for indicating an entity number. An elegant alternative is to prefix such variables with an *i*: `iTable, iEmployee`. This effectively makes them *named* iterators.

9. Iterator variables should be called ***i***, ***j***, ***k*** etc.

```
while (Iterator i = pointList.iterator(); i.hasNext(); ) {
  :
}

for (int i = 0; i < nTables; i++) {
  :
}
```

The notation is taken from mathematics where it is an established convention for indicating iterators. Variables named *j*, *k* etc. should be used for nested loops only.

10. Complement names must be used for complement entities.

get/set, add/remove, create/destroy, start/stop, insert/delete, increment/decrement, old/new, begin/end, first/last, up/down, min/max, next/previous, old/new, open/close, show/hide
Reduce complexity by symmetry.

11. Abbreviations in names should be avoided.

```
computeAverage();      // NOT:  compAvg();
```

There are two types of words to consider. First are the common words listed in a language dictionary. These must never be abbreviated. Never write:

`cmd`	instead of	`command`
`cp`	instead of	`copy`
`pt`	instead of	`point`
`comp`	instead of	`compute`
`init`	instead of	`initialize`

etc.

Then there are domain specific phrases that are more naturally known through their acronym or abbreviations. These phrases should be kept abbreviated. Never write:

`HypertextMarkupLanguage`	instead of	`html`
`CentralProcessingUnit`	instead of	`cpu`
`PriceEarningRatio`	instead of	`pe`

etc.

12. Negated boolean variable names must be avoided.

```
boolean isError;     // NOT:   isNotError
boolean isFound;     // NOT:   isNotFound
```

The problem arise when the logical **NOT** operator is used and double negative arises. It is not immediately apparent what `!isNotError` means.

13. Functions (methods returning an object) should be named after what they return and procedures (*void* methods) after what they do. This increases readability. Makes it clear what the unit should do and especially all the things it is *not* supposed to do. This again makes it easier to keep the code clean of side effects. Naming pointers in C++ specifically should be clear and should represent the pointer type distinctly.

```
Line *line          //NOT    Line *pLine; or Line *lineptr; etc
```

Lecture No. 29

10.4 File handling tips for Java and C++

1. C++ header files should have the extension *.h*. Source files can have the extension *.c++* (recommended), *.C*, *.cc* or *.cpp*.

```
MyClass.c++, MyClass.h
```

These are all accepted C++ standards for file extension.

2. Classes should be declared in individual header files with the file name matching the class name. Secondary private classes can be declared as inner classes and reside in the file of the class they belong to. All definitions should reside in source files.

```
class MyClass
{
  public:
    int getValue () {return value_;} // NO!
    ...
  private:
    int value_;
}
```

The header files should declare an interface, the source file should implement it. When looking for an implementation, the programmer should always know that it is found in the source file. The obvious exception to this rule is of course inline functions that must be defined in the header file.

3. Special characters like TAB and page break must be avoided. These characters are bound to cause problem for editors, printers, terminal emulators or debuggers when used in a multi-programmer, multi-platform environment.

4. The incompleteness of split lines must be made obvious.

```
totalSum = a + b + c +
                  d + e);
    function (param1, param2,
                param3);
     setText ("Long line split"
                      "into two parts.");
    for (tableNo = 0; tableNo < maxTable;
                     tableNo += tableStep)
```

Split lines occurs when a statement exceed the 80 column limit given above. It is difficult to give rigid rules for how lines should be split, but the examples above should give a general hint.

In general:

- Break after a comma.
- Break after an operator.
- Align the new line with the beginning of the expression on the previous line.

Include Files and Include Statements for Java and C++

1. Header files must include a construction that prevents multiple inclusion. The convention is an all uppercase construction of the module name, the file name and the h suffix.

```
#ifndef MOD_FILENAME_H
#define MOD_FILENAME_H
 :
#endif
```

The construction is to avoid compilation errors. The construction should appear in the top of the file (before the file header) so file parsing is aborted immediately and compilation time is reduced.

Classes and Interfaces

Class and Interface declarations should be organized in the following manner:

1. Class/Interface documentation.
2. `class` or `interface` statement.
3. Class (**`static`**) variables in the order **`public`**, **`protected`**, package (no access modifier), **`private`**.
4. Instance variables in the order **`public`**, **`protected`**, package (no access modifier), `private`.
5. Constructors.
6. Methods (no specific order).

10.5 Statements in Java and C++

Types

1. Type conversions must always be done explicitly. Never rely on implicit type conversion.

```
floatValue = (float) intValue;     // NOT:    floatValue = intValue;
```

By this, the programmer indicates that he is aware of the different types involved and that the mix is intentional.

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

3. The parts of a class must be sorted *public*, *protected* and *private*. All sections must be identified explicitly. Not applicable sections should be left out. The ordering is *"most public first"* so people who only wish to use the class can stop reading when they reach the `protected/private` sections.

Variables

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

2. Variables must never have dual meaning. This enhances readability by ensuring all concepts are represented uniquely. Reduce chance of error by side effects.

3. Class variables should never be declared public. The concept of information hiding and encapsulation is violated by public variables. Use private variables and access functions instead. One exception to this rule is when the class is essentially a data structure, with no behavior (equivalent to a C++ `struct`). In this case it is appropriate to make the class' instance variables public.

4. Related variables of the same type can be declared in a common statement. Unrelated variables should not be declared in the same statement.

```
float                 x,                 y,                 z;
float  revenueJanuary, revenueFebrury, revenueMarch;
```

The common requirement of having declarations on separate lines is not useful in the situations like the ones above. It enhances readability to group variables.

5. Variables should be kept alive for as short a time as possible. Keeping the operations on a variable within a small scope, it is easier to control the effects and side effects of the variable.

6. Global variables should not be used. Variables should be declared only within scope of their use. Same is recommended for global functions or file scope variables. It is easier to control the effects and side effects of the variables if used in limited scope.

7. Implicit test for *0* should not be used other than for boolean variables and pointers.

```
if (nLines != 0)     // NOT:   if (nLines)
if (value != 0.0)    // NOT:   if (value)
```

It is not necessarily defined by the compiler that ints and floats 0 are implemented as binary 0. Also, by using explicit test the statement give immediate clue of the type being tested. It is common also to suggest that pointers shouldn't test implicit for 0 either, i.e. `if (line == 0)` instead of `if (line)`. The latter is regarded as such a common practice in C/C++ however that it can be used.

Loop structures

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

```
sum = 0;                    // NOT:     for (i=0, sum=0; i<100; i++)
for (i=0; i<100; i++)       //             sum += value[i];
  sum += value[i];
```

2. Loop variables should be initialized immediately before the loop.

```
boolean done = false;  // NOT:   boolean  done = false;
while (!done) {        //          :
  :                    //        while (!done) {
}                      //          :
                                  }
```

3. The use of *do .... while* loops should be avoided. There are two reasons for this. First is that the construct is superflous; Any statement that can be written as a *do .... while* loop can equally well be written as a *while* lopp or a *for* loop. Complexity is reduced by minimizing the number of constructs being used. The other reason is of readability. A loop with the conditional part at the end is more difficult to read than one with the conditional at the top.

4. The use of *break* and *continue* in loops should be avoided. These statements should only be used if they prove to give higher readability than their structured counterparts. In general *break* should only be used in *case* statements and *continue* should be avoided alltogether.

5. The form *for (;;)* should be used for empty loops.

```
for (;;) {             // NOT:   while (true) {
  :                    //          :
}                      //        }
```

This form is better than the functionally equivalent *while (true)* since this implies a test against *true*, which is neither necessary nor meaningful. The form while(true) should be used for infinite loops.

Conditionals

1. Complex conditional expressions must be avoided. Introduce temporary boolean variables instead.

```
if ((elementNo < 0) || (elementNo > maxElement)||
    elementNo == lastElement) {
  :
}
```

should be replaced by:

```
boolean isFinished      = (elementNo < 0) || (elementNo > maxElement);
boolean isRepeatedEntry = elementNo == lastElement;
if (isFinished || isRepeatedEntry) {
  :
}
```

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

```
boolean isError = readFile (fileName);
if (!isError) {
  :
}
else {
  :
}
```

3. The conditional should be put on a separate line.

```
if (isDone)               // NOT:  if (isDone) doCleanup();
  doCleanup();
```

4. Executable statements in conditionals must be avoided.

```
file = openFile (fileName, "w");  // NOT:   if ((file = openFile
(fileName, "w")) != null) {
if (file != null) {               //          :
  :                               //        }
}
```

Miscellaneous

1. The use of magic numbers in the code should be avoided. Numbers other than *0* and *1* should be considered declared as named constants instead.

2. Floating point constants should always be written with decimal point and at least one decimal.

```
double total = 0.0;    // NOT: double total = 0;
double speed = 3.0e8; // NOT: double speed = 3e8;

double sum;
:
sum = (a + b) * 10.0;
```

This emphasizes the different nature of integer and floating point numbers even if their values might happen to be the same in a specific case. Also, as in the last example above, it emphasize the type of the assigned variable (`sum`) at a point in the code where this might not be evident.

3. Floating point constants should always be written with a digit before the decimal point.

```
double total = 0.5;    // NOT: double total = .5;
```

The number and expression system in Java is borrowed from mathematics and one should adhere to mathematical conventions for syntax wherever possible. Also, 0.5 is a lot more readable than .5; There is no way it can be mixed with the integer 5.

4. Functions in C++ must always have the return value explicitly listed.

```
int getValue()      // NOT: getValue()
{
  :
}
```

If not explicitly listed, C++ implies `int` return value for functions.

5. `goto` in C++ should not be used. Goto statements violates the idea of structured code. Only in some very few cases (for instance breaking out of deeply nested structures) should goto be considered, and only if the alternative structured counterpart is proven to be less readable.

Lecture No. 30

10.6 Layout and Comments in Java and C++

Comments

The problem with comments is that they lie. Comments are not syntax checked, there is nothing forcing them to be accurate. And so, as the code undergoes change during schedule crunches, the comments become less and less accurate.

As Fowler puts it, comments should not be used as deodorants. Tricky code should not be commented but rewritten. In general, the use of comments should be minimized by making the code self-documenting by appropriate name choices and an explicit logical structure.

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

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

2. Use // for all comments, including multi-line comments.

```
// Comment spanning
// more than one line
```

Since multilevel commenting is not supported in C++ and Java, using // comments ensure that it is always possible to comment out entire sections of a file using /* */ for debugging purposes etc.

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

```
while (true) {          // NOT:     while (true) {
  // Do something       //          // Do something
  something();          //              something();
}                       //          }
```

10.7 Expressions and Statements

Layout

1. Basic indentation should be 2.

```
for (i = 0; i < nElements; i++)
  a[i] = 0;
```

Indentation of 1 is to small to emphasize the logical layout of the code. Indentation larger than 4 makes deeply nested code difficult to read and increase the chance that the lines must be split. Choosing between indentation of 2, 3 and 4, 2 and 4 are the more common, and 2 chosen to reduce the chance of splitting code lines.

Natural form for expression

Expression should be written as if they are written as comments or spoken out aloud. Conditional expression with negation are always difficult to understand. As an example consider the following code:

if (! (block < activeBlock) || !(blockId >= unblocks))

The logic becomes much easier to follow if the code is written in the natural form as shown below:

if ((block >= activeBlock) || (blockId < unblocks))

Parenthesize to remove ambiguity

Parentheses should always be used as they reduce complexity and clarify things by specifying grouping. It is especially important to use parentheses when different unrelated operators are used in the same expression as the precedence rules are often assumed by the programmers, resulting in logical errors that are very difficult to spot. As an example consider the following statement:

if (x & MASK == BITS)

This causes problems because == operator has higher precedence than & operator. Hence, MASK and BITS are first compared for equality and then the result, which is 0 or 1, is andded with x. This kind of error will be extremely hard to catch. If, however, parentheses are used, there will be no ambiguity as shown below.

if ((x & MASK) == BITS)

Following is another example of the use of parentheses which makes the code easier to understand and hence easier to maintain.

```
leapYear = year % 4 == 0  && year % 100 != 0 || year % 400 == 0 ;
```

In this case parentheses have not been used and therefore the definition of a leap year is not very clear for the code. The code becomes self explanatory with the help of proper use of parentheses as shown below:

```
leapYear = ((year % 4 == 0)  && (year % 100 != 0))  ||
                    (year % 400 == 0);
```

Breakup complex expressions

Complex expressions should be broken down into multiple statements. An expression is considered to be complex if it uses many operators in a single statement. As an example consider the following statement:

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

This statement liberally uses a number of operators and hence is very difficult to follow and understand. If it is broken down into simple set of statements, the logic becomes easier to follow as shown below:

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

10.8 Shortcuts and cryptic code

Sometimes the programmers, in their creative excitement, try to write very concise code by using shortcuts and playing certain kinds of tricks. This results in a code which is cryptic in nature and hence is difficult to follow. Maintenance of such code therefore becomes a nightmare. Following are some examples of such code.

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

   ```
   x *= a;
   ```

   For some reason the code was later modified to:

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

This seemingly harmless change is actually a little cryptic and causes confusion. The problem lies with the semantics of this statement. Does it mean x = x*a+b or x = x*(a+b)? The second one is the right answer but is not obvious from the syntax and hence causes problems.

2. Let us now look at a more complex example. What is the following code doing?

```
subkey = subkey >> (bitoff – (bitoff >> 3) << 3));
```

As can be seen, it is pretty hard to understand and therefore difficult to debug in case there is any problem. What this code is actually doing is masking bitoff with octal 7 and then use the result to shift subkey those many time. This can be written as follows:

```
subkey = subkey >> (bitoff & 0x7);
```

It is quite evident that the second piece of code is much follow to read than the first one.

3. The following piece of code is taken from a commercial software:

```
a = a >> 2;
```

It is easy to see that a is shifted right two times. However, the real semantics of this code are hidden – the real intent here is to divide a by 4. No doubt that the code above achieves this objective but it is hard for the reader to understand the intent as to why a is being shifted right twice. It would have been much better had the code been written as follows:

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

4. A piece of code similar to the following can be found in many data structures books and is part of circular implementation of queues using arrays.

```
bool Queue::add(int n)
{
        int k = (rear+1) % MAX_SIZE;
        if (front == k)
                return false;
        else {
                rear = k;
                queue[rear] = n;
                return true;
        }
}
```

```
bool Queue::isFull()
{
        if (front == (rear+1 % MAX_SIZE))
                return true;
        else
                return false;
}
```

This code uses the % operator to set the rear pointer to 0 once it has reached MAX_SIZE. This is not obvious immediately. Similarly, the check for queue full is also not trivial.

In an experiment, the students were asked to implement double-ended queue in which pointer could move in both directions. Almost everyone made the mistake of writing something like rear = (rear-1) % MAX_SIZE. This is because the semantics of % operation are not obvious.

It is always much better to state the logic explicitly. Also, counting is also much easier to understand and code as compared to some tricky comparisons (e.g. check for isFull in this case). Application of both these principles resulted in the following code. It is easy to see that this code is easier to understand. It is interesting to note that when another group of students were asked to do implement double-ended queue after showing them this code, almost everyone did it without any problems.

```
bool Queue::add()
{
        if (! isFull() ) {
                rear++;
                if (rear == MAX_SIZE) rear = 0;
                QueueArray[rear] = n;
                size++;
                return true;
        }
        else return false;
}

bool Queue::isFull(int n)
{
        if (size == MAX_SIZE)
                return true;
        else
                return false;
}
```

Lecture No. 31

Coding Style Guidelines (Continued)

Switch Statement

In the switch statement, cases should always end with a break. A tricky sequence of fall-through code like the one below causes more trouble than being helpful.

```
switch(c) {
case '-'  : sign = -1;
case '+'  : c = getchar();
case '.'  : break;
default   : if (! isdigit(c))
                        return 0;
}
```

This code is cryptic and difficult to read. It is much better to explicitly write what is happening, even at the cost of duplication.

```
switch(c) {
case '-':   sign = -1;
                c = getchar();
                break;
case '+':   c = getchar();
                break;
case '.':   break;
default:    if (! isdigit(c))
                    return 0;
                break;
}
```

It would even be better if such code is written using the if statement as shown below.

```
if (c == '-') {
        sign = -1;
        c = getchar();
}
else if (c == '+') {
        c = getchar();
}
else if (c != '.' && !isdigit(c)) {
        return 0;
}
```

Magic Numbers

Consider the following code segment:

```
fac = lim / 20;
if (fac < 1)
        fac = 1;
for (i =0, col = 0; i < 27; i++, j++) {
        col += 3;
        k = 21 – (let[i] /fac);
        star = (let[i] == 0) ? ‘ ’ : ‘*’;
        for (j = k; j < 22; j++)
                draw(j, col, star);
}
draw(23, 1, ‘ ’);
for (i=‘A’; i <= ‘Z’; i++)
        cout << i;
```

Can you tell by reading the code what is meant by the numbers 20, 27, 3, 21, 22, and 23. These are constant that mean something but they do not give any indication of their importance or derivation, making the program hard to understand and modify. To a reader they work like magic and hence are called magic numbers. Any number (even 0 or 1) used in the code is a magic number. It should rather have a name of its own that can be used in the program instead of the number.

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

```
enum {
        MINROW = 1,
        MINCOL = 1,
        MAXROW = 24,
        MAXCOL = 80,
        LABELROW = 1,
        NLET = 26,
        HEIGHT = MAXROW –4,
        WIDTH = (MAXCOL-1) / NLET
};

fac = (lim+HEIGHT-1) /HEIGHT;
if (fac < 1)
        fac = 1;
for (i =0; i < NLET; i++) {
        if (let[i] == 0)
                continue;
        for (j = HEIGHT – let[i] / fac; j < HEIGHT; j++)
                draw(j-1 + LABELROW,
                        (i+1)*WIDTH, ‘*’);
}

draw(MAXROW-1, MINCOL+1, ‘ ’);
```

```
for (i='A'; i <= 'Z'; i++)
        cout << i;
```

Use (or abuse) of Zero

The number 0 is the most abused symbol in programs written in C or C++. One can easily find code segment that 0 in a fashion similar to the examples below in almost every C/C++ program.

```
flag = 0;          // flag is boolean
str = 0;           // str is string
name[i] = 0;       // name is char array
x = 0;             // x is floating pt
i = 0;             // i is integer
```

This is a legacy of old style C programming. It is much better to use symbols to explicitly indicate the intent of the statement. It is easy to see that the following code is more in line with the self-documentation philosophy than the code above.

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

Lecture No. 32

10.9 Clarity through modularity

As mentioned earlier, abstraction and encapsulation are two important tools that can help in managing and mastering the complexity of a program. We also discussed that the size of individual functions plays a significant role in making the program easy or difficult to understand. In general, as the function becomes longer in size, it becomes more difficult to understand. Modularity is a tool that can help us in reducing the size of individual functions, making them more readable. As an example, consider the following selection sort function.

```
void selectionSort(int a[], int size)
{
        int i, j;
        int temp;
        int min;

        for (i = 0; i < size-1; i++){
                min = i;
                for (j = i+1; j < size; j++){
                        if (a[j] < a[min])
                                min = j;
                }
                temp = a[i];
                a[i] = a[min];
                a[min] = temp;
        }
}
```

Although it is not very long but we can still improve its readability by breaking it into small functions to perform the logical steps. The modified code is written below:

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

int minimum(int a[], int from, int to)
{
        int i;
        int min;
        min = a[from];
        for (i = from; i <= to; i++){
                if (a[i] < a[min])
                        min = i;
        }
        return min;
}

void selectionSort(int a[], int size)
{
        int min;
        int i;
        for (i = 0; i < size; i++){
                min = minimum(a, i, size –1);
                swap(a[i], a[min]);
        }
}
```

It is easy to see that the new selectionSort function is much more readable. The logical steps have been abstracted out into the two functions namely, minimum and swap. This code is not only shorter but also as a by product we now have two functions (minimum and swap) that can be reused.

Reusability is one of the prime reasons to make functions but is not the only reason. Modularity is of equal concern (if not more) and a function should be broken into smaller pieces, even if those pieces are not reused. As an example, let us consider the quickSort algorithm below.

```
void quickSort(int a[], int left, int right)
{
        int i, j;
        int pivot;
        int temp;
        int mid = (left + right)/2;
        if (left < right){
                i = left - 1;
                j = right + 1;
                pivot = a[mid];
                do {
                        do i++; while (a[i] < pivot);
                        do j--; while (a[i] < pivot);
                        if (i<j){
                                temp = a[i];
                                a[i] = a[j];
                                a[j] = temp;
                        }
                } while (i < j);
                temp = a[left];
                a[left] = a[j];
                a[j] = temp;

                quickSort(a, left, j);
                quickSort(a, j+1, right);
}
```

This is actually a very simple algorithm but students find it very difficult to remember. If is broken in logical steps as shown below, it becomes trivial.

```
void quickSort(int a[], int left, int right)
{
        int p;
        if (left < right){
                p = partition(a, left, right);
                quickSort(a, left, p-1);
                quickSort(a, p+1, right);
        }
}


int partition(int a[], int left, int right)
{
        int i; j;
        int pivot;
        i = left + 1;
        j = right;
        pivot = a[left];
        while(i < right && a[i] < pivot) i++;
        while(j > left && a[j] >= pivot) j++;
        if(i < j)
                swap(a[i], a[j]);
        swap(a[left], a[j]);
        return j;
}
```

10.10 **Short circuiting || and &&**

The logical and operator, &&, and logical or operators, ||, are special due to the C/C++ short circuiting rule, i.e. a || b and a && b are short circuit evaluated. That is, logical expressions are evaluated left to right and evaluation stops as soon as the final truth value can be determined.

Short-circuiting is a very useful tool. It can be used where one boolean expression can be placed first to "guard" a potentially unsafe operation in a second boolean expression. Also, time is saved in evaluation of complex expressions using operators || and &&. However, a number of issues arise if proper attention is not paid.

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

```
struct Node {
        int data;
        Node * next;
};


Node *ptr;
...

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

What's wrong with this code?

The second part of condition, ptr != NULL, is supposed to be the guard. That is, if the value of the pointer is NULL, then the control should not enter the body of the while loop otherwise, it should check whether ptr->data < myData or not and then proceed accordingly. When the guard is misplaced, if the pointer is NULL then the program will crash because it is illegal to access a component of a non-existent object. This code is rewritten as follows. This time the short-circuiting helps in achieving the desired objective which would have been a little difficult to code without such help.

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

10.11 Operand Evaluation Order and Side Effects

A side effect of a function occurs when the function, besides returning a value, changes either one of its parameters or a variable declared outside the function but is accessible to it. That is, a side effect is caused by an operation that may return an explicit result but it may also modify the values stored in other data objects. Side effects are a major source of programming errors and they make things difficult during maintenance or debugging activities. Many languages do not specify the function evaluation order in a single statement. This combined with side effects causes major problems. As an example, consider the following statement:

```
c = f1(a) + f2(b);
```

The question is, which function (f1 or f2) will be evaluated first as the C/C++ language does not specify the evaluation order and the implementer (compiler writer) is free to choose one order or the other. The question is: does it matter?

To understand this, let's look at the definition of f1 and f2.

```
int f1(int &x)
{
        x = x * 2;
        return x + 1;
}

int f2(int &y)
{
        y = y / 2;
        return y - 1;
}
```

In this case both f1 and f2 have side effects as they both are doing two things - changing the value of the parameter and changing the value at the caller side. Now if we have the following code segment,

```
a = 3;
b = 4;

c = f1(a) + f2(b);
```

then the value of a, b, and c would be as follows:

```
a = 6
b = 2
c = 8
```

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 when i is incremented.

3.
“,” is very dangerous as it causes side effects. Let’s look at the following statement:

```
int i, j = 0;
```
Because of the syntax, many people would assume that i is also being initialized to 0, while it is not. Combination of , and = -- is fatal. Look at the following statement:

```
a = b, c = 0;
```
A majority of the programmers would assume that all a, b, and c are being initialized to 0 while only c is initialized and a and b have garbage values in them. This kind of overlook causes major programming errors which are not caught easily and are caused only because there are side effects.

Guidelines

If the following guidelines are observed, one can avoid hazards caused by side effects.

1. never use “,” except for declaration
2. if you are initializing a variable at the time of declaration, do not declare another variable in the same statement
3. never use multiple assignments in the same statement
4. Be very careful when you use functions with side effects – functions that change the values of the parameters.
5. Try to avoid functions that change the value of some parameters and return some value at the same time.

10.12 Performance

In many cases, performance and maintainability are at odds with one another. When planning for performance, one should always remember the 80/20 rule - you spend 80 percent of your time in 20 percent of the code. That is, we should not try to optimize everything. The proper approach is to profile the program and then identify bottlenecks to be optimized. This is similar to what we do in databases – we usually normalize the database to remove redundancies but then partially de-normalize if there are performance issues.

As an example, consider the following. In this example a function isspam is profiled by calling 10000 times. The results are shown in the following table:

sec	%	Cum%	cycles	instructions	calls	function
			990000	10000	00	strchr
			80000	6000	00	strncmp
			0000	0000	00	strstr
			5559	2213	35	strlen
			000	000		isspam
11 other functions with insignificant performance overhead						

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

When a small set (a couple of functions) of functions which use each other is so overwhelmingly the bottleneck, there are two alternatives:

1. use a better algorithm
2. rewrite the whole set

In this particular case strstr was rewritten and profiled again. It was and found out that although it was much faster but now 99.8% of the time was spent in strstr.

The algorithm was rewritten and restructured again by eliminating strstr, strchr, and strncmp and used memcmp. Now memcmp was much more complex than strstr but it gained efficiency by eliminating a number of loops and the new results are as shown:

sec	%	Cum%	cycles	instructions	calls	function
			0000	90000	000	memcmp
			0000	0000		isspam
			4	3		strlen

strlen now went from over two million calls to 652.

Many details of the execution can be discovered by examining the numbers. The trick is to concentrate on hot spots by first identifying them and then cooling them. As mentioned earlier, most of the time is spent in loops. Therefore we need to concentrate on loops.

As an example, consider the following:

```
for (j = i; j < MAX_FIELD; j++)
  clear(j);
```

This loop clears field before each new input is read. It was observed that it was taking almost 50% of the total time. On further investigation it was found out that MAX_FIELD was 200 but the actual fields that needed to be cleared were 2 or 3 in most cases. The code was subsequently modified as shown below:

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

This reduced the overall execution time by half.

Lecture No. 34

10.13 Portability

Many applications need to be ported on to many different platforms. As we have seen, it is pretty hard to write error free, efficient, and maintainable software. So, if a major rework is required to port a program written for one environment to another, it will be probably not come at a low cost. So, we ought to find ways and means by which we can port applications to other platforms with minimum effort. The key to this lies in how we write our program. If we are careful during writing code, we can make it portable. On the other hand if we write code without portability in mind, we may end-up with a code that is extremely hard to port to other environment. Following is brief guideline that can help you in writing portable code.

Stick to the standard

1. Use ANSI/ISO standard C++
2. Instead of using vendor specific language extensions, use STL as much as possible

Program in the mainstream

Although C++ standard does not require function prototypes, one should always write them.

```
double sqrt();          // old style acceptable by ANSI C

double sqrt(double);    // ANSI – the right approach
```

Size of data types

Sizes of data types cause major portability issues as they vary from one machine to the other so one should be careful with them.

```
int i, j, k;

…
j = 20000;
k = 30000;

i = j + k;
```

```
                    // 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 implementation to other. This therefore also causes portability issues. We should therefore follow guidelines mentioned in the side effect discussion.

Signedness of char

The language does not specify whether char is signed or unsigned.

```
char c;
        // between 0 and 255 if unsigned
        // -128 to 127 if signed

c = getchar();
if (c == EOF)  ??

        // will fail if it is unsigned
```

It should therefore be written as follows:

```
int c;
c = getchar();
if (c == EOF)
```

Arithmetic or Logical Shift

The C/C++ language has not specified whether right shift >> is arithmetic or logical. In the arithmetic shift sign bit is copied while the logical shift fills the vacated bits with 0. This obviously reduces portability.

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

Byte Order and Data Exchange

The order in which bytes of one word are stored is hardware dependent. For example in Intel architecture the lowest byte is the most significant byte while in Motorola architecture the highest byte of a word is the most significant one. This causes problem when dealing with binary data and we need to be careful while exchanging data between to heterogeneous machines. One should therefore only use text for data exchange. One should also be aware of the internationalization issues and hence should not assume ASCII as well as English.

Alignment

The C/C++ language does not define the alignment of items within structures, classes, an unions. data may be aligned on word or byte boundaries. For example:

```
struct X {
        char c;
        int i;
};
```

address of i could be 2, 4, or 8 from the beginning of the structure. Therefore, using pointers and then typecasting them to access individual components will cause all sorts of problems.

Bit Fields

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

- Packing several objects into a machine word. *e.g.* 1 bit flags can be compacted -- Symbol tables in compilers.
- Reading external file formats -- non-standard file formats could be read in. *E.g.* 9 bit integers.

C lets us do this in a structure definition by putting :*bit length* after the variable. *i.e.*

```
struct packed_struct {
        unsigned int f1:1;
        unsigned int f2:1;
        unsigned int f3:1;
        unsigned int f4:1;
        unsigned int type:4;
        unsigned int funny_int:9;
} pack;
```

Here the `packed_struct contains 6 members: Four 1 bit` *flags* `f1..f3, a 4 bit type and a 9 bit funny_int.`

C automatically packs the above bit fields as compactly as possible, provided that the maximum length of the field is less than or equal to the integer word length of the computer. If this is not the case then some compilers may allow memory overlap for the fields whilst other would store the next field in the next word.

Bit fields are a convenient way to express many difficult operations. However, bit fields do suffer from a lack of portability between platforms:

- integers may be signed or unsigned
- Many compilers limit the maximum number of bits in the bit field to the size of an `integer` which may be either 16-bit or 32-bit varieties.
- Some bit field members are stored left to right others are stored right to left in memory.
- If bit fields too large, next bit field may be stored consecutively in memory (overlapping the boundary between memory locations) or in the next word of memory.

Bit fields therefore should not be used.

Lecture No. 35

10.14 Exception handling

Exception handling is a powerful technique that separates error-handling code from normal code. It also provides a consistent error handling mechanism. The greatest advantage of exception handling is its ability to handle asynchronous errors.

The idea is to raise some error flag every time something goes wrong. There is a system that is always on the lookout for this error flag. Third, the previous system calls the error handling code if the error flag has been spotted. The raising of the imaginary error flag is simply called *raising* or *throwing* an error. When an error is thrown the overall system responds by *catching* the error. Surrounding a block of error-sensitive code with exception handling is called *trying* to execute a block. The following code segment illustrates the general exception handling mechanism.

```
try {
___...
___...
___throw Exception()
___...
___...
} catch( Exception e )
{
___...
___...
}
```

One of the most powerful features of exception handling is that an error can be thrown over function boundaries. This allows programmers to put the error handling code in one place, such as the *main*-function of your program.

Exceptions and code complexity

A number of invisible execution paths can exist in simple code in a language that allows exceptions. The complexity of a program may increase significantly if there are exceptional paths in it. Consider the following code:

```
String EvaluateSalaryAnadReturnName( Employee e)
{
  if (e.Title() == "CEO" || e.Salary() > 10000)
  {
        cout << e.First() << " " << e.Last() << " is overpaid" << endl;
  }
  return e.First() + " " + e.Last();
}
```

Before moving any further, let's take the following assumptions:

1. Different order of evaluating function parameters are ignored.
2. Failed destructors are ignored
3. Called functions are considered atomic
    a. for example: e.Title() could throw for several reasons but all that matters for this function is whether e.Title() results in an exception or not.
4. To count as different execution paths, an execution path must be made-up of a unique sequence of function calls performed and exited in the same way.

Question: How many more execution paths are there?

Ans: 23. There are 3 non-exceptional paths and 20 exceptional paths.

The non-exceptional paths:

The three non-exceptional paths are enumerated as below:

1. if (e.Title() == "CEO" || e.Salary() > 10000)
    - if e.Title() == "CEO" is true then the second part is not evaluated and e.Salary() will not be called.
    - cout will be performed

2. if e.Title() != "CEO" and e.Salary() > 10000
    - both parts of the condition will be evaluated
    - cout will be performed.

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

Exceptional Code Paths

The 20 exceptional code path are listed below.

1. String EvaluateSalaryAnadReturnName(Employee e)

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

2. if (e.Title() == “CEO” || e.Salary() > 10000)

   e.Title() might itself throw, or it might return an object of class type by value, and that copy operation might throw.

3. if (e.Title() == “CEO” || e.Salary() > 10000)

   Same as above.

4. if (e.Title() == “CEO” || e.Salary() > 10000)

   To match a valid ==() operator, the string literal may need to be converted to a temporary object of class type and that construction of the temporary might throw.

5. if (e.Title() == “CEO” || e.Salary() > 10000)

   Same as above.

6. if (e.Title() == “CEO” || e.Salary() > 10000)

   operator ==() might throw.

7. if (e.Title() == “CEO” || e.Salary() > 10000)

   Same as above.

8. if (e.Title() == “CEO” || e.Salary() > 10000)

Same as above.

9-13 cout << e.First() << “ “ << e.Last() << “ is overpaid” << endl;

As per C++ standard, any of the five calls to << operator might throw.

14-15 cout << e.First() << “ “ << e.Last() << “ is overpaid” << endl

similar to 2 and 3.

16-17 return e.First() + “ “ + e.Last();

similar to 14-15.

18-19 return e.First() + “ “ + e.Last();

similar to 6,7, and 8.

20. return e.First() + “ “ + e.Last();

similar to 4.

Summary:

- A number of invisible execution paths can exist in simple code in a language that allows exceptions.
- Always be exception-aware. Know what code might emit exceptions.

The Challenge:
Can we make this code exception safe and exception neutral? That is, rewrite it (if needed) so that it works properly in the presence of an exception and propagates all exceptions to the caller?

Exception-Safety:
A function is exception safe if it might throw but do not have any side effects if it does throw and any objects being used, including temporaries, are exception safe and clean-up there resources when destroyed.

Exception Neutral:
A function is said to be exception neutral if it propagates all exceptions to the caller.

Levels of Exception Safety

- Basic Guarantee: Ensures that temporaries are destroyed properly and there are no memory leaks.
- Strong Guarantee: Ensures basic guarantee as well as there is full-commit or roll-back.
- No-throw Guarantee: Ensure that a function will not throw.

Exception-safety requires either no-throw guarantee or basic and strong guarantee.

Does the function satisfy basic guarantee?

Yes. Since the function does not create any objects, in the presence of an exception, it does not leak any resources.

Does the function satisfy strong guarantee?

No. The strong guarantee says that if the function fails because of an exception, the program state must not be change.

This function has two distinct side-effects:

- an overpaid message is emitted to cout.
- A name strings is returned.

As far as the second side-effect is concerned, the function meets the strong guarantee because if an exception occurs the value will never be returned.

As far as the first side-effect is concerned, the function is not exception safe for two reasons:

- if exception is thrown after the first part of the message has been emitted to cout but before the message has been completed (for example if the fourth << operator throws), then a partial message was emitted to cout.
- If the message emitted successfully but an exception occurs in later in the function (for example during the assembly of the return value), then a

message was emitted to cout even though the function failed because of an exception. It should be complete commit or complete roll-back.

Does the function satisfy no-throw guarantee?

No. This is clearly not true as lots of operations in the function might throw.

Strong Guarantee

To meet strong guarantee, either both side-effects are completed or an exception is thrown and neither effect is performed.

```
// First attempt:

String EvaluateSalaryAnadReturnName( Employee e)
{
        String result = e.First() + " " + e.Last();

        if (e.Title() == "CEO" || e.Salary() > 10000)
        {

                String message = result + " is overpaid\n";
                cout << message;

        }
        return result;
}
```

What happens if the function is called as follows:

```
String theName;
theName = evaluateSalarayAndReturnName(someEmplyee);
```

1. string copy constructor is invoked because the result is returned by value.
2. The copy assignment operator is invoked to copy the result into theName
3. If either copy fails then the function has completed its side-effects (since the message was completely emitted and the return value was completely constructed) but the result has been irretrievable lost.

Can we do better and perhaps avoid the problem by avoiding the copy?

```
// Second attempt:

String EvaluateSalaryAnadReturnName( Employee e, String &r)
{
        String result = e.First() + " " + e.Last();

        if (e.Title() == "CEO" || e.Salary() > 10000)
        {

                String message = result + " is overpaid\n";
                cout << message;

        }
        r = result;
}
```

Looks better but assignment to r might still fail which leaves us with one side-effect completed and other incomplete.

```
// Third attempt:

auto_ptr<String> EvaluateSalaryAnadReturnName( Employee e)
{
        auto_ptr<String> result = new String(e.First() + " " + e.Last());

        if (e.Title() == "CEO" || e.Salary() > 10000)
        {

                String message = (*result) + " is overpaid\n";
                cout << message;

        }
        return result;();       // rely on transfer of ownership
                                // this can't throw
}
```

We have effectively hidden all the work to construct the second side-effect (the return value), while we ensured that it can be safely returned to the caller using only non-throwing operation after the first side-effect has completed the printing of the message. In this case we know that once the function is complete, the return value will make successfully into the hands of the caller and be correctly cleaned-up in all cases. This is because the aut_ptr semantics guarantee that If the caller accepts the returned value, the act of accepting a copy of the auto_ptr causes the caller to take the ownership and if the caller does not accept the returned value, say by ignoring the

return value, the allocated string will automatically be destroyed with proper clean-up.

Exception Safety and Multiple Side-effects

It is difficult and some-times impossible to provide strong exception safety when there are two or more side-effects in one function and these side-effects are not related with each other. For example we would not have been able to provide exception safety if there are two output messages, one to cout and the other one to cerr. This is because the two cannot be combined.

When such a situation comes with two or more unrelated side-effects which cannot be combined then the best way to handle such a situation is break it into two separate functions. That way, at least, the caller would know that these are two separate atomic steps.

Summary

1. Providing the strong exception-safety guarantee often requires you to trade-off performance.
2. If a function has multiple un-related side-effects, it cannot always be made strongly exception safe. If not, it can be done only by splitting the function into several functions, each of whose side-effects can be performed atomically.
3. Not all functions need to be strongly exception-safe. Both the original code and attempt#1 satisfy the basic guarantee. For many clients, attempt # 1 is sufficient and minimizes the opportunity for side-effects to occur in the exceptional situation, without requiring the performance trade-off of attempt #3.

Lecture No. 36

Software Verification and Validation

11.1 Software Testing

To understand the concept of software testing correctly, we need to understand a few related concepts.

Software verification and validation

Verification and validation are the processes in which we check a product against its specifications and the expectations of the users who will be using it. According to a known software engineering expert Berry Boehm, verification and validation are

Verification

- Does the product meet system specifications?
- Have you built the product right?

Validation

- Does the product meet user expectations?
- Have you built the right product?

It is possible that a software application may fulfill its specifications but it may deviate from users expectations or their desired behavior. That means, software is verified but not validated. How is it possible? It is possible because during the requirements engineering phase, user needs might not have been captured precisely or the analyst might have missed a major stakeholder in the analysis. Therefore, it is important to verify as well as validate the software product.

11.2 Defect

The second major and a very important concept is Defect. A defect is a variance from a desired product attribute. These attributes may involve system specifications well as user expectation. Anything that may cause customer dissatisfaction, is a defect. Whether these defects are in system specifications or in the software products, it is essential to point these out and fix.

Therefore software defect is that phenomenon in which software deviates from its expected behavior. This is non-compliance from the expected behavior with respect to written specifications or the stakeholder needs.

Software and Defect

Software and defects go side by side in the software development life cycle. According to a famous saying by Haliburton, Death and taxes are inevitable. According to Kernighan: Death, taxes, and bugs are the only certainties in the life of a programmer. Software and defects cannot be separated, however, it is important to learn how discovering defects at an appropriate stage improves the software quality. Therefore, software application needs to be verified as well as validated for a successful deployment.

Software Testing

With these concepts, we are in a position to define software testing. Software testing is the process of examining the software product against its requirements. Thus it is a process that involves verification of product with respect to its written requirements and conformance of requirements with user needs. From another perspective, software testing

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

Software testing objective

- The correct approach to testing a scientific theory is not to try to verify it, but to seek to refute the theory. That is to prove that it has errors. (Popper 1965)
- The goal of testing is to expose latent defects in a software system before it is put to use.
- A software tester tries to break the system. The objective is to show the presence of a defect not the absence of it.
- Testing cannot show the absence of a defect. It only increases your confidence in the software.
- This is because exhaustive testing of software is not possible – it is simply too expansive and needs virtually infinite resources.

Successful Test

From the following sayings, a successful test can be defined

"If you think your task is to find problems then you will look harder for them than if you think your task is to verify that the program has none" – Myers 1979.

"A test is said to be successful if it discovers an error" – doctor's analogy.

The success of a test depends upon the ability to discover a bug not in the ability to prove that the software does not have one. As, it is impossible to check all the different scenarios of a software application, however, we can apply techniques that can discover potential bugs from the application. Thus a test that helps in discovering a bug is a successful test. In software testing phase, our emphasis is on discovering all the major bugs that can be identified by running certain test scenarios. However it is important to keep in mind that testing activity has certain limitations.

Limitations of testing

With the help of the following example, we shall see how difficult it may become to discover a defect from a software application.

Example (adapted from Backhouse)

This is a function that compares two strings of characters stored in an array for equality. The tester who has to test this function has devised the following test cases in which different combinations of inputs are tried and their expected behavior is documented. From the look of this table, it seems that almost all major combinations of inputs have been documented.

Inputs and Expected Outputs

A	b	Expected result
"cat"	"dog"	False
""	""	True
"hen"	"hen"	True
"hen"	"heN"	False
" "	""	False

“”	“ball”	False
“cat”	“”	False
“HEN”	“hen”	False
“rat”	“door”	False
“ ”	“ ”	True

Results of testing

The tester runs all the above-mentioned test cases and function returns the same results as expected. But it is still not correct. What can be a problem To analyze the problem, lets look at the code of this string equal routine

Code of the Function

```
bool isStringsEqual(char a[], char b[])
{
   bool result;
   if (strlen(a) != strlen(b))
      result = false;
   else {
      for (int i =0; i < strlen(a); i++)
         if (a[i] == b[i])
            result = true;
         else result = false;
   }
   return result;
}
```

Analysis of the code

It passes all the designated tests but fails for two different strings of same length ending with the same character. For example, “cut” and “rat” would results in true which is not correct.

The above-mentioned defect signifies a clear limitation of the testing process in discovering a defect which is not very frequent. However, it should be noted from this example that a tester cannot generate all possible combinations of test cases to test an application as the number of scenarios may become exhaustive.

Testing limitations

- In order to prove that a formula or hypothesis is incorrect all you have to do to show only one example in which you prove that the formula or theorem is not working.
- On the other hand, million of examples can be developed to support the hypothesis but this will not prove that it is correct.
- These examples only help you in coming up with a hypothesis but they are not proves by themselves and they only enhance your comfort level in that particular hypothesis or in this particular case, in your piece of software.
- You cannot test a program completely because:
  - the domain of the possible inputs is too large to test.
  - there are too many possible paths through the program to test.
- According to Discrete Mathematics

- To prove that a formula or hypothesis is incorrect you have to show only one example.
- To prove that it is correct any numbers of examples are insufficient. You have to give a formal proof of its correctness.

11.3 Test Cases and Test Data

In order to test a software application, it is necessary to generate test cases and test data which is used in the application. Test cases correspond to application functionality such that the tester writes down steps which should be followed to achieve certain functionality. Thus a test case involves

- Input and output specification plus a statement of the function under test.
- Steps to perform the function
- Expected results that the software application produces

However, test data includes inputs that have been devised to test the system.

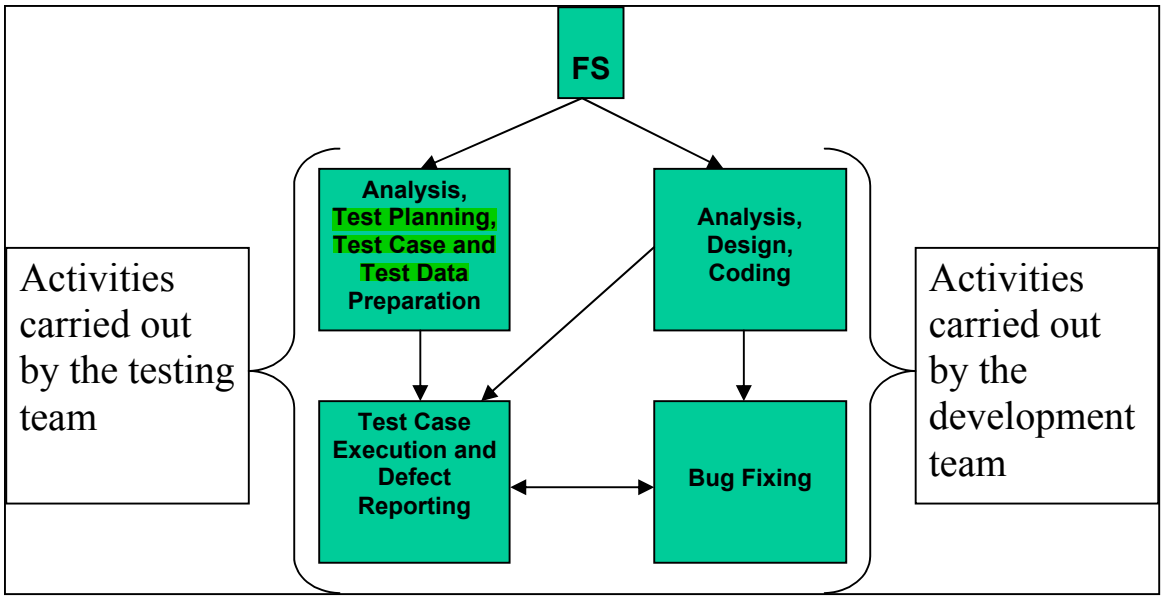
Lecture No. 37

11.3 Testing vs. development

Testing is an intellectually demanding activity and has a lifecycle parallel to software development. A common misperception about testing is that it is not a challenging activity. It should be noted here that the testing demands grip over the domain and application functionality as is required from an analyst or a designer who has to develop the application from requirements. As without having an in-depth knowledge about the system and the requirements from users, a tester cannot write test cases that can verify and validate software application with respect to documented specifications and user needs. Writing test cases and generating test data are processes that demand scenario-building capabilities. These activities essentially require destructive instincts in a tester for the purpose of breaking system to discover loopholes into its functionality.

At the time when these two activities are being performed, merely initial design of the application is completed. Therefore, tester uses his/her imagination to come up with use patterns of the application that can help him/her in describing exact steps that should be executed in order to test a particular functionality. Moreover, tester needs to figure out loose points in the system from where he/she can discover defects. All these activities are highly imaginative and a tester is supposed to possess above average (if not excellent) analytical skills.

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



Description

- Functional specification document is the starting point, base document for both testing and the development
- Right side boxes describe the development, whereas, left side boxes explain the testing process
- Development team is involved into the analysis, design and coding activities.
- Whereas, testing team too is busy in analysis of requirements, for test planning, test cases and test data generation.
- System comes into testing after development is completed.
- Test cases are executed with test data and actual results (application behavior) are compared with the expected results,
- Upon discovering defects, tester generates the bug report and sends it to the development team for fixing.
- Development team runs the scenario as described in the bug report and try to reproduce the defect.
- If the defect is reproduced in the development environment, the development team identifies the root cause, fixes it and sends the patch to the testing team along with a bug resolution report.
- Testing team incorporates the fix (checking in), runs the same test case/scenario again and verifies the fix.
- If problem does not appear again testing team closes down the defect, otherwise, it is reported again.

11.5 The Developer and Tester

Development	Testing
Development is a creative activity	Testing is a destructive activity
Objective of development is to show that the program works	Objective of testing is to show that the program does not work

Scenarios missed or misunderstood during development analysis would never be tested correctly because the corresponding test cases would either be missing or would be incorrect.

The left side column is related to the development, and the right side describes the testing. Development is a creative process as developers have to build the system, whereas, testing is a destructive activity as the goal of a tester is to break the system to discover the defects. Objective of development is to show that the program works, objective of testing is to show that program does not work. However, FS is the base document for both of these activities.

Tester analyzes FS with respect to testing the system whereas; developer analyzes FS with respect to designing and coding the system. If developer does not understand the FS correctly then he cannot implement and test it right. Thus if the same person who has developed a system, tests it, chances of carrying the same misunderstanding in testing will be very high. Therefore, an independent testing can only prove his understanding wrong. Therefore, it is highly recommended that developer should not try to test his/her own work.

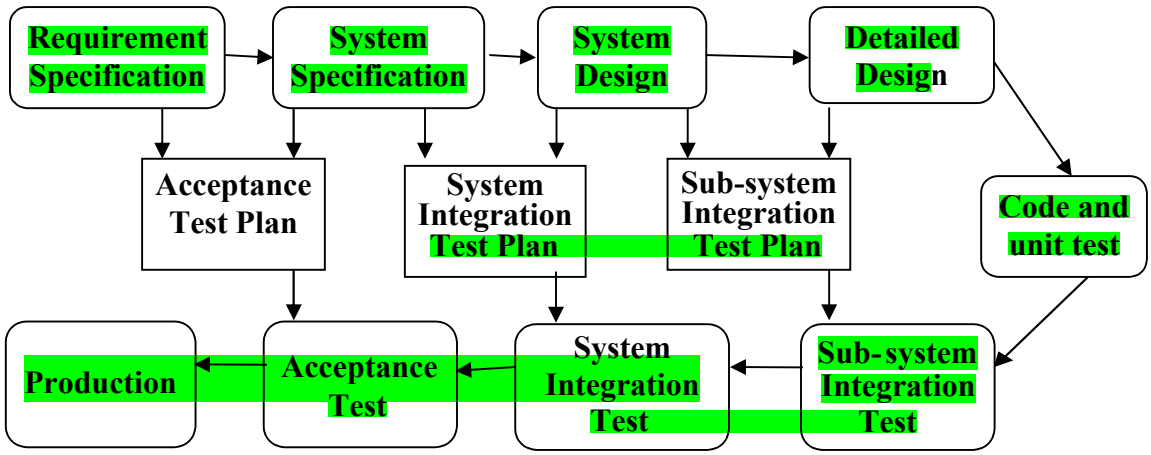
11.6 Usefulness of testing

Objective of testing is to discover and fix as many errors as possible before the software is put to use. That is before it is shipped to the client and the client runs it for acceptance. In software development organizations, a rift exists between the development and the testing teams. Often developers are found questioning about the significance or even need to have the testing resources in the project teams. Whoever doubts on the usefulness of the testing team should understand what could happen if the application is delivered to client without testing? At the best, the client may ask to fix all the defects (free of cost) he would discover during the acceptance testing. At the worst, probably he would sue the development firm for damages. However, in practice, clients are often seen complaining about the deliverables and a couple of defected deliverables are sufficient for breaking the relations next to the cancellation of contract.

Therefore, it should be well preserved among the community of developers that testers are essential rather inevitable. A good tester has a knack of smelling errors – just like auditors and it is for the good of the organization not to harm it.

11.7 Testing and software phases

With the help of the following diagram we shall explain different phases of testing



Software development process diagram

Description of testing phases

- Unit testing – testing individual components independent of other components.
- Module testing – testing a collection of dependent components – a module encapsulates related components so it can be tested independently.
- Subsystem testing – testing of collection of modules to discover interfacing problems among interacting modules.
- System testing – after integrating subsystems into a system – testing this system as a whole.
- Acceptance test – validation against user expectations. Usually it is done at the client premises.
- Alpha testing – acceptance testing for customized projects, in-house testing for products.
- Beta testing – field testing of product with potential customers who agree to use it and report problem before system is released for general use

In the following two types of testing activities are discussed.

11.8 Black box testing

In this type of testing, a component or system is treated as a black box and it is tested for the required behavior. This type of testing is not concerned with how the inputs are transformed into outputs. As the system's internal implementation details are not visible to the tester. He gives inputs using an interface that the system provides and tests the output. If the outputs match with the expected results, system is fine otherwise a defect is found.

11.9 Structural testing (white box)

As opposed to black box testing, in structural or white box testing we look inside the system and evaluate what it consists of and how is it implemented. The inner of a system consists of design, structure of code and its documentation etc. Therefore, in white box testing we analyze these internal structures of the program and devise test cases that can test these structures.

Effective testing

The objective of testing is to discover the maximum number of defects with a minimum number of resources before the system is delivered to the next stage. Now the question arises here how to increase the probability of finding a defect?

As, good testing involves much more than just running the program a few times to see whether it works or not. A good tester carries out a thorough analysis of the program to devise test cases that can be used to test the system systematically and effectively. Problem here is how to develop a representative set of test cases that could test a complete program. That is, selection of a few test cases from a huge set of possibilities. What should be the sets of inputs that should be used to test the system effectively and efficiently?

String Equal Example

- For how many equal strings do I have to test to be in the comfortable zone?
- For how many unequal strings do I have to test to be in the comfortable zone?

- When should I say that further testing is unlikely to discover another error? Testing types

To answer these questions, we divide a problem domain in different classes. These are called Equivalence Classes.

Lecture No. 38

Equivalence Classes or Equivalence Partitioning

Two tests are considered to be equivalent if it is believed that:

- if one discovers a defect, the other probably will too, and
- if one does not discover a defect, the other probably won't either.

Equivalence classes help you in designing test cases to test the system effectively and efficiently. One should have reasons to believe that the test cases are equivalent. As for this purpose, one would need to understand the system and see in how many partitions it can be divided. These partitions should be devised such that a clear distinction should be marked. Test cases written for one partition should not yield the same results when run for the second partition as otherwise these two partitions should become one. However, finding equivalence classes is a subjective process, as two people analyzing a program will probably come up with different sets of equivalence classes .

Equivalence partitioning guidelines

- Organize your equivalence classes. Write them in some order, use some template, sequence, or group them based on their similarities or distinctions. These partitions can be hierarchical or organized in any other manner.
- Boundary conditions: determine boundary conditions. For example, adding in an empty linked list, adding after the last element, adding before the first element, etc.
- You should not forget invalid inputs that a user can give to a system. For example, widgets on a GUI, numeric instead of alphabets, etc.

Equivalence partitioning example

In the following example, we shall see how equivalence partitions can be developed for a string matching function.

String matching

Organization

For equivalence partitions, we divide the problem in two obvious categories of equal strings and the other one for unequal strings. Within these equivalent partitions, further partitioning is done. Following is the description of the equivalence partitions and their test cases

Test cases for equivalence partitions

Equal

- Two equal strings of arbitrary length
  - All lower case "cat" "cat"
  - All upper case "CAT" "CAT"

- Mixed case "Cat" "Cat"
- Numeric values "123" "123"
- Two strings with blanks only " " " "
- Numeric and character mixed "Cat1" "Cat1"
- Strings with special characters "Cat#1" "Cat#1"

- Two NULL strings "" ""

Unequal Strings

- Two different equal strings of arbitrary length
  - Two strings with different length "cat" "mouse"
  - Two strings of same length "cat" "dog"
- Check for case sensitivity
  - Same strings with different characters capitalized "Cat" "caT"
- One string is empty
  - First is NULL "" "cat"
  - Second is NULL "cat" ""

11.10 Basis Code Structures

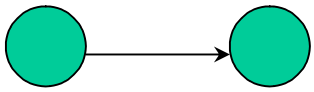
For structural testing it is important to know about basic coding structures. There are four basic coding structures sequence, if statement, case statement, and while loop. These four basic structures can be used to express any type of code.

Flow graph notation

In analysis and design, you have already seen the flow graph notation. This is used to describe flow of data or control in an application. However, we do not use flow graphs to describe decisions. That is, how a branch is taken is not shown in flow graphs.
In the following, we are using flow graph notation to describe different coding structures.

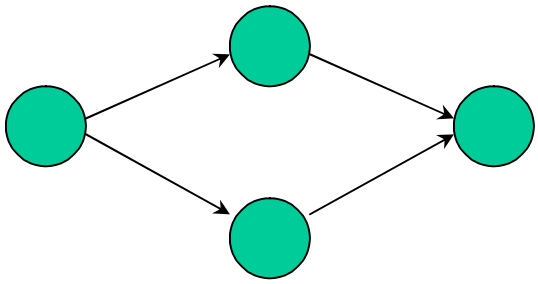
Sequence

Sequence depicts programming instructions that do not have branching or any control information. So we lump together several sequential instructions in one node of the graph.



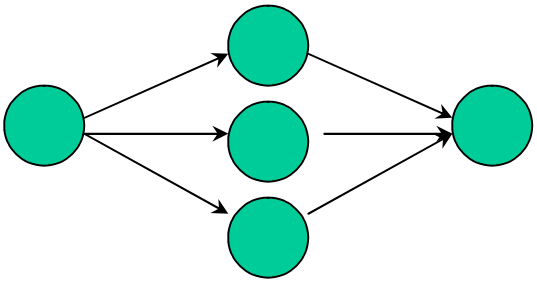
If

Second structural form is the If statement. In the following graph, the first node at the left depicts the if statement and the two nodes next to the first node correspond to the successful case (if condition is true) and unsuccessful case (if condition is false) consecutively. The control comes to the same instruction from either of these intermediate instructions.



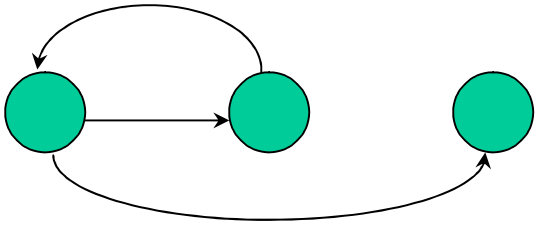
Case

In Case statement, control can take either of several branches (as opposed to only two in If statement.) First node represents the switch statement (C/C++) and nodes in middle correspond to all different cases. Program can take one branch and result into the same instruction.



While

A while loop structure consists of a loop guard instruction through which the iteration in the loop is controlled. The control keeps iterating in the loop as long as the loop guard condition is true. It branches to the last instruction when it becomes false.



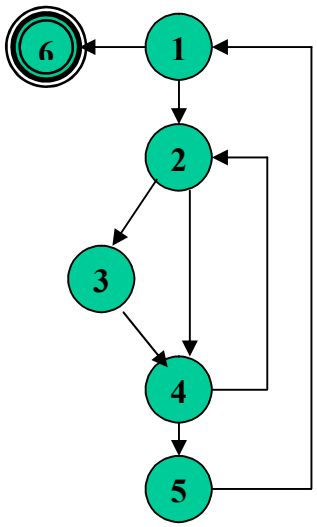
Flow graph for bubble sort

In the following example, code is given for a bubble sort function. The diagram opposite to the code is the corresponding flow graph model. It consists of six nodes. Node one is the while loop instruction that contains loop guard and node six is the ending instruction of the while loop. Node two corresponds to the for loop instructions. Node three corresponds to the swapping instruction. Nodes five and six correspond to the last instructions of the if statement and the for loop consecutively.

Point to note here is the assignment of node numbers to program instructions. As, the corresponding flow graph model would consist of nodes that correspond to instructions which are major decision points in the code. For example, when this function will be invoked, control will certainly come to the while loop instruction and at the minimum it will traverse from node one to node six. However, if control enters into the while loop even for a single iteration, it will traverse through nodes two, four and five for certain and node three too if it enters for loop and check in the if condition is true.

So all these combinations are to be tested during white box testing.

```
sorted = false;
while (!sorted) {                  //1
    sorted = true;
    for (i=0; i < N-1; i++) {      //2
        if a[i] > a[i+1] {
            swap(a[i], a[i+1]);    //3
            sorted = false;
        }                          //4
    }                              //5
}                                  //6
```



Paths

Following are possible paths from starting to the end of this code.
Path1: 1-6
Path2: 1-2-3-4-5-1-6
Path3: 1-2-4-5-1-6
Path4: 1-2-4-2-3-4-5-6-1

Lecture No. 39

White box testing

As described in the section above, in white box testing we test the structure of the program. In this technique the test cases are written in a manner to cover different possibilities in code. Below are described three coverage schemes.

Coverage

- **Statement Coverage**: In this scheme, statements of the code are tested for a successful test that checks all the statements lying on the path of a successful scenario.
- **Branch Coverage**: In this scheme, all the possible branches of decision structures are tested. Therefore, sequences of statements following a decision are tested.
- **Path Coverage**: In path coverage, all possible paths of a program from input instruction to the output instruction are tested. An exhaustive list of test cases is generated and tested against the code.

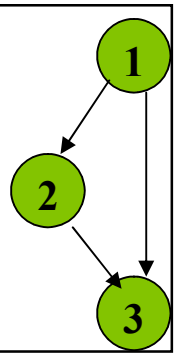
White Box Testing Example

With the help of the following example, we shall see how many test cases are generated for each type of coverage schemes.

```
if (a = = b)          //1

    c = d;            //2

a++;                  //3
```



Statement coverage:

- a=1,b=1
- If a==b then statement 2, statement 3

Branch coverage

- Statement 1: two braches 1-2, 1-3
- Test case 1: if a =1, b=1 then statement 2
- Test case 2: if a=1,b=2: then statement 3

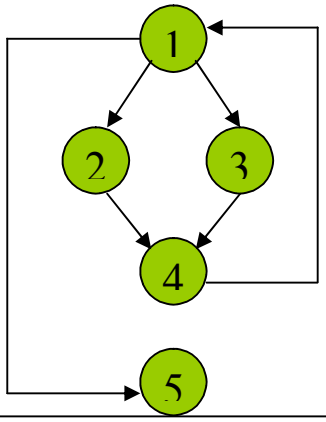
Path coverage

Same as branch testing

Paths in a program containing loops

Now we shall apply the path coverage scheme on a piece of code that contains a loop statement and see how many test cases can possibly be developed.

```
for (i = 0; i < N; i++) {          //1
  if (condition1)
        // do something here       //2
  else
        // do something here       //3
  // something here                //4
}
                                   //5
```



Paths

The following is an analysis of the above-mentioned code and the flow diagram. It determines the number of paths against different iterations of the loop.

- N = 0: If the control does not enter into the loop then only one path will be traversed. It is 1-5.
- N=1: Two different paths can possibly be traversed (depending on condition).
  - 1-2-4-1-5
  - 1-3-4-1-5
- N=2: Four possible paths can be traversed.
  - 1-2-4-1-2-4-1-5
  - 1-2-4-1-3-4-1-5

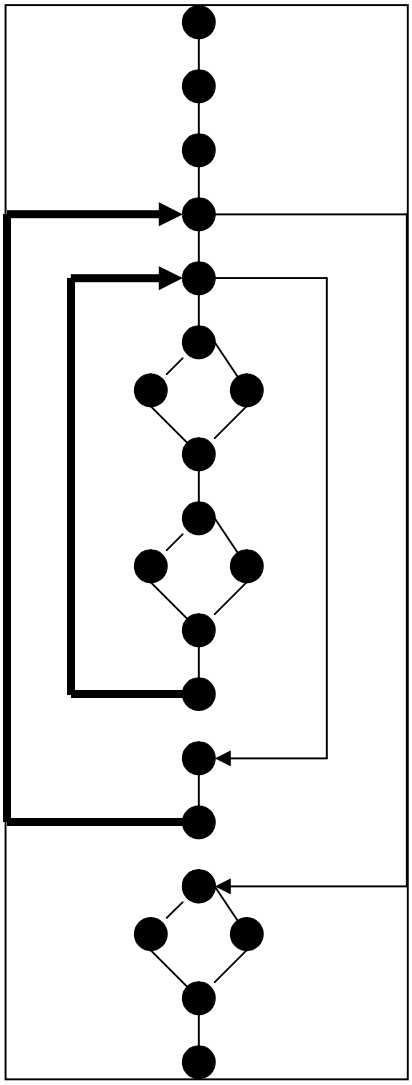
- 1-3-4-1-2-4-1-5
- 1-3-4-1-3-4-1-5

- Generalizing the relation between loop variable N and the number of possible paths, for the value of N, 2^N paths are possible
  - Thus if N = 20 it means more then 1 million paths are possible.

Flow graph of a hypothetical program

Following is a flow graph of a hypothetical program whose structure is such that it consists of two loops, one embedded in the other. Left side of the diagram signifies the loops where inner loop seem to be containing two if statements. Second loop has a branch and then program finishes.

Simple graph contains 1852 paths with each loop not iterated more than twice



Thus, the number of paths in a program that contains loops tends to infinity. It is impossible to conduct exhaustive testing of a program that may consist of infinite number of test cases. The question arises, how many test cases need to be executed in order to test all the major scenarios in the code at least once? The answer is, calculate the cyclomatic

complexity of the code. This will give us a number that corresponds to the total number of test cases that need to be generated in order to test all the statements and branches in the code at least once.

Cyclomatic complexity

The concept of cyclomatic complexity is extremely useful in white box testing when analyzing the relative complexity of the program to be tested. It revolves around independent paths in a program which is any path through the program (from start to end) that introduces at least one new set of processing statements or a new condition. An independent path covers statements and branches of the code.

Cyclomatic complexity is a quantitative measure of the logical complexity of a program. It defines number of independent paths in the basis set of a program. It provides an upper bound for the number of tests that must be conducted to ensure that all statements and branches have been executed at least once.

Cyclomatic Complexity, V(G), for a flow graph G is defined as:

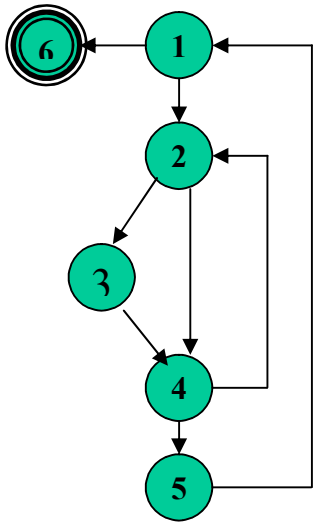
$$V(G) = E - N + 2$$

Where E is the number of edges and N is the number of nodes in the flow graph G. Cyclomatic complexity provides us with an upper bound for the number of independent paths that comprise the basis set.

Cyclomatic Complexity of a Sort Procedure

Following is the same bubble sort program that we discussed above. This time we shall calculate its cyclomatic complexity and see how many test cases are needed to test this function.

```
Sorted = FALSE;
while (!sorted) {                  //1
   sorted = TRUE;
   for (i=0; i < N-1; i++) {       //2
      if a[i] > a[i+1] {
         swap(a[i], a[i+1]);       //3
         sorted = FALSE;
      }                            //4
   }                               //5
}                                  //6
```



Cyclomatic complexity

- Number of edges = 8
- Number of nodes = 6
- C(G) = 8-6+2 = 4

Paths to be tested

- Path1: 1-6

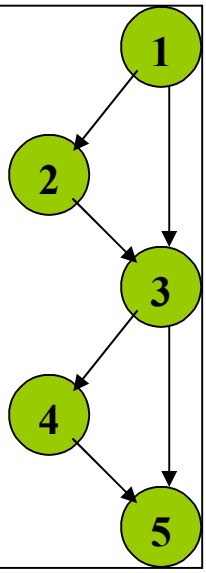
- Path2: 1-2-3-4-5-1-6
- Path3: 1-2-4-5-1-6
- Path4: 1-2-4-2-3-4-5-6-1

Infeasible paths

Infeasible path is a path through a program which is never traversed for any input data.

Example

```
if (a == b)          //1
    c = c-1;         //2
if (a != b)          //3
    c = c+1;         //4
                     //5
```



In the above-mentioned example, there are two infeasible paths that will never be traversed.

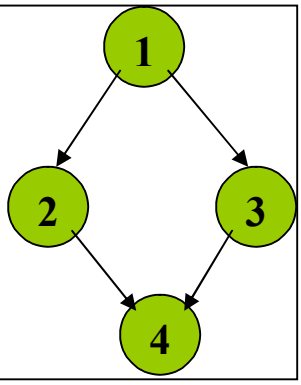
- Path1: 1-2-3-4-5
- Path2: 1-3-5

A good programming practice is such that minimize infeasible paths to zero. It will reduce the number of test cases that need to be generated in order to test the application. How can we minimize infeasible paths, by simply using else part with the if statement and avoid program statements as given above.

Modified code segment

Infeasible paths can be analyzed and fixed.

```
if (a == b)          //1
      c = c-1;       //2
else
      c = c+1;       //3
                     //4
```



There are no infeasible paths now!

Lecture No. 40

11.11 Unit testing

A software program is made up of units that include procedures, functions, classes etc. The unit testing process involves the developer in testing of these units. Unit testing is roughly equivalent to chip-level testing for hardware in which each chip is tested thoroughly after manufacturing. Similarly, unit testing is done to each module, in isolation, to verify its behaviour. Typically the unit test will establish some sort of artificial environment and then invoke routines in the module being tested. It then checks the results returned against either some known value or against the results from previous runs of the same test (regression testing). When the modules are assembled we can use the same tests to test the system as a whole.

Software should be tested more like hardware, with

- Built-in self testing: such that each unit can be tested independently
- Internal diagnostics: diagnostics for program units should be defined.
- Test harness

The emphasis is on built in testability of the program units from the very beginning where each piece should be tested thoroughly before trying to wire them together.

Unit Testing Principles

- In unit testing, developers test their own code units (modules, classes, etc.) during implementation.
- Normal and boundary inputs against expected results are tested.
- Thus unit testing is a great way to test an API.

Quantitative Benefits

- **Repeatable**: Unit test cases can be repeated to verify that no unintended side effects have occurred due to some modification in the code.
- **Bounded**: Narrow focus simplifies finding and fixing defects.
- **Cheaper**: Find and fix defects early

Qualitative Benefits

- **Assessment-oriented**: Writing the unit test forces us to deal with design issues - cohesion, coupling.
- **Confidence-building**: We know what works at an early stage. Also easier to change when it's easy to retest.

Testing against the contract (Example)

When we write unit tests we want to write test cases that ensure a given unit honors its contract. This will tell us whether the code meets the contract and whether the contract means what we think the unit is supposed to do in the program.

Contract for square root routine

```
result = squareRoot(argument);
assert (abs (result * result – argument) < epsilon);
```

The above contract tells us what to test:

- Pass in a negative argument and ensure that it is rejected
- Pass in an argument of zero to ensure that it is accepted (this is a boundary value)
- Pass in values between zero and the maximum expressible argument and verify that the difference between the square of the result and the original argument is less than some value epsilon.

When you design a module or even a single routine, you should design both its contract and the code to test that contract. By designing code to pass a test that fulfill its contract, you might consider boundary conditions and other issues that you wouldn't consider otherwise. The best way to fix errors is to avoid them in the first place. By building the tests *before* you implement the code you get to try out the interface before you commit to it.

Unit Testing Tips

Unit test should be conveniently located

- For small projects you can imbed the unit test for a module in the module itself
- For larger projects you should keep the tests in the package directory or a /test subdirectory of the package

By making the code accessible to developers you provide them with:

- Examples of how to use all the functionality of your module
- A means to build regression tests to validate any future changes to the code

You can use the main routine with conditional compilation to run your unit tests.

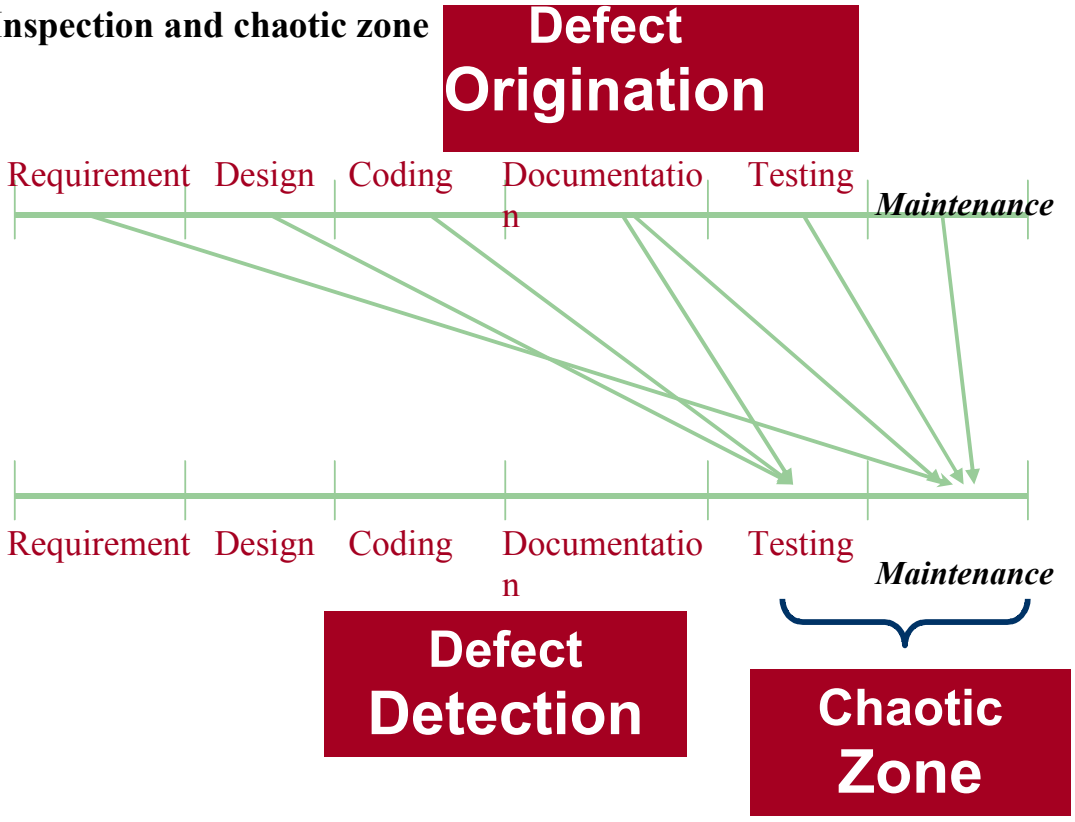
11.12 Defect removal efficiency

Is the ability to remove defects from the application. We shall further elaborate this idea with the help of the above-mentioned table.

Design Inspection					●		●	●	●
Code Inspection				●		●		●	●
Quality Assurance			●			●	●		●
Testing		●	●	●	●	●	●	●	●
Worst	**30%**	**37%**	**50%**	**55%**	**65%**	**75%**	**77%**	**85%**	**95%**
Median	**40%**	**53%**	**65%**	**70%**	**80%**	**87%**	**90%**	**97%**	**99%**
Best	**50%**	**60%**	**75%**	**80%**	**87%**	**93%**	**95%**	**99%**	**99.9%**

The above table depicts data that was published after analyzing 1500 projects. In these projects, four different types of quality assurance mechanisms were employed. It is evident from this table that testing alone can only remove 53% of defects. However, testing and quality assurance mechanisms combine yield up to 65% efficiency. Whereas, if we combine code inspection and testing together, results are up to 75%. Similarly, design inspections and testing yield up to 80%. Moreover, combining design inspections, quality assurance and testing results are up to 95%. If all four techniques are combined, results are up to 99.9%.

Inspection and chaotic zone



In this diagram, a chaotic zone has been defined. In fact, if defects are not discovered and fixed at the appropriate stage, then at the testing and maintenance phases, these defects are piled up. Therefore, a chaotic zone is formed for the testing and the development teams as the number of defects which are piled up, destabilize the application and it becomes extremely hard to fix all these defects as some of these bugs may involve changes in requirements and design. At testing or the maintenance phases, fixing a defect in requirements or design becomes extremely expensive, as underlying code will have to be changed as well.

If we combine the results of the above two diagrams, it is evident that testing alone does not suffice. We need to employ inspection techniques and combine them with testing to increase the effectiveness of defect removal efficiency.

11.13 Defect origination

In inspections the emphasis is on early detection and fixing of defects from the program. Following are the points in a development life cycle where defects enter into the program.

- Requirements
- Design
- Coding
- User documentation
- Testing itself can cause defects due to bad fixes
- Change requests at the maintenance or initial usage time

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

Lecture No. 41

11.14 Inspection versus Testing

Inspections and testing are complementary and not opposing verification techniques. Both should be used during the verification and validation process. Inspections can check conformance with a specification but not conformance with the customer's real requirements. Inspections cannot check non-functional characteristics such as performance, usability, etc. Inspection does not require execution of program and they maybe used before implementation. Many different defects may be discovered in a single inspection. In testing, one defect may mask another so several executions are required. For inspections, checklists are prepared that contain information regarding defects. Reuse domain and programming knowledge of the viewers likely to help in preparing these checklists. Inspections involve people examining the source representation with the aim of discovering anomalies and defects. Inspections may be applied to any representation of the system (requirements, design, test data, etc.) Thus inspections are a very effective technique for discovering errors in a software program.

Inspection pre-conditions

A precise specification must be available before inspections. Team members must be familiar with the organization standards. In addition to it, syntactically correct code must be available to the inspectors. Inspectors should prepare a checklist that can help them during the inspection process.

Inspection checklists.

Checklist of common errors in a program should be developed and used to drive the inspection process. These error checklists are programming language dependent such that the inspector has to analyze major constructs of the programming language and develop checklists to verify code that is written using these checklists. For example, in a language of weak type checking, one can expect a number of peculiarities in code that should be verified. So the corresponding checklist can be larger. Other example of programming language dependant defects are defects in variable initialization, constant naming, loop termination, array bounds, etc.

Inspection Checklist

Following is an example of an inspection checklist.

Exception management faults	• Have all possible error conditions been taken into account?
Fault Class	• Inspection Check

Data faults	• Are all program variables initialized before their values are used? • Have all constants been named? • Should the lower bound of arrays be 0, 1, or something else? • Should the upper bound of arrays be size or size -1? • If character strings are used, is a delimiter explicitly assigned?
Control faults	• For each conditional statement, is the condition correct? • Is each loop certain to terminate? • Are compound statements correctly bracketed? • In case statements, are all possible cases accounted for?
Input/Output faults	• Are all input variables used? • Are all output variables assigned a value before they are output?
Interface faults	• Do all function and procedure calls have correct number of parameters? • Do formal and actual parameters types match? • Are the parameters in right order? • If components access shared memory, do they have the same model of shared memory structure?
Storage management faults	• If a linked structure is modified, have all links been correctly assigned? • If dynamic storage is used, has space been allocated correctly? • Is space explicitly de-allocated after it is no longer required?

In the checklist mentioned above, a number of fault classes have been specified and their corresponding inspection checks are described in the column at the right side. This type of checklist helps an inspector to look for specific defects in the program. These inspection checks are the outcomes of experience that the inspector has gained out of developing or testing similar programs.

11.15 Static analyzers

Static analyzers are software tools for source text processing. They parse the program text and try to discover potentially erroneous conditions and bring these to the attention of the verification and validation team. These tools are very effective as an aid to inspections. But these are supplement to but not a replacement for inspections.

Checklist for static analysis

Data faults	• variable used before initialization • variable declared but never used • variables assigned twice but never used between assignments • possible array bound violations • undeclared variables
Control faults	• unreachable code • unconditional branches into loops
Input/Output faults	• variable output twice with no intervening assignment
Storage Management fault	• unassigned pointers • pointer arithmetic

Lecture No. 42

Debugging

12.1 Debugging

As Benjamin Franklin said, "in this world, nothing is certain but death and taxes." If you are in the software development business, however, you can amend that statement. Nothing in life is certain except death, taxes, and software bugs. If you cryogenically freeze yourself, you can delay death indefinitely. If you move to a country with no income tax, you can avoid paying taxes by not buying anything. If you develop software, however, no remedy known to mankind can save you from the horror of software bugs.

What is a Bug?

We call them by many names: software defects, software bugs, software problems, and even software "features." Whatever you want to call them, they are things the software does that it is not supposed to do (or, alternatively, something the software doesn't do that it is supposed to). Software bugs range from program crashes to returning incorrect information to having garbled displays.

12.2 A Brief History of Debugging

It would appear that as long as there have been computers, there have been computer bugs. However, this is not exactly true. Even though the earliest known computer programs contained errors, they were not, at that time, referred to as "bugs." It took a lady named Admiral Grace Hopper to actually coin the term "bug."

After graduating from Vassar in 1928, she went to Yale to receive her master's degree in mathematics. After graduating from Yale, she worked at the university as a mathematics professor. Leaving Yale in 1943, with the onset of World War II, Mrs. Hopper decided to work for the Navy. Mrs. Hopper's first assignment was under Commander Howard Aiken at Howard University, working at the Bureau of Ordinance Computation. She was a programmer on the Mark II, the world's first automatically sequenced digital computer. The Mark II was used to determine shooting angles for the big guns in varying weather conditions during wartime.

It was during her term with the Mark II that Hopper was credited with coining the term "bug" for a computer problem. The first "bug" was actually a moth, which flew through an open window and into one of the Mark II's relays. At that time, physical relays were used in computers, unlike digital components we use today. The moth shorted out across two contacts, temporarily shutting down the system. The moth would later be removed (de-bugged?) and pasted into the logbook of the project. From that point on, if her team was not producing numbers or working on the code, they claimed to be "debugging" the system.

From that auspicious beginning, computer debugging developed into something of a hit-or-miss procedure for quite a few years. Early debugging efforts mostly centered around either data dumps of the system or used output devices, such as printers and display lights, to indicate when an error occurred. Programmers would then step through the code line by line until they could determine the location of the problem. The next step in the

evolution of debugging came with the advent of command-line debuggers. These simple programs were an amazing step forward for the programmers. Although difficult to use, even at the time, these programs represented the first real attempt to turn debugging from a hit-or-miss proposition into a reproducible process.
Once the debugger became a part of the programmer's arsenal, the software world began to consider other ways that programs could be more easily debugged. Software projects starting getting bigger, and the same techniques that worked well for small projects no longer worked when the program reached a certain size.

Importance of Debugging

As we mentioned earlier in this course, one of the prime objectives of software engineering is to develop cost effective software applications. According to a survey, when a software application is in the maintenance phase, 20% of its lifecycle cost is attributed towards the defects which are found in the software application after installation. Please bear in mind that the maintenance is the phase in which 2/3rd of the overall software cost incurs. Therefore, 20% of the 2/3rd cost is again a huge cost and we need to understand why this much cost and effort is incurred. In fact, when a software application is installed and being used, any peculiarity in it can cost a lot of direct and indirect damages to the organization. A system downtime is the period in which tremendous pressure is on developers end to fix the problem and make the system running again. In these moments, every second costs hugs losses to the organization and it becomes vital to find out the bug in the software application and fix it. Debugging techniques are the only mechanism to reach at the code that is malfunctioning. In the following subsection, we shall discuss an incident that took place in 1990 and see how much loss the company had to suffer due to a mere bug in the software application.

12.4 Problem at AT&T

In the telecommunications industry, loss of service is known as an outage. For most of us, when as outage occurs, we lose our telephone service; we cannot make calls and we cannot receive calls. Outages are accepted and expected hazards in the industry.
On January 15. 1990, AT&T had a US wide telephone system outage that lasted for nine hours. The cause was due to a program error in the software that was meant to make the system more efficient. Eight years later, on April 13, 1998, AT&T suffered another massive failure in its frame-relay network, which affected ATM machines, credit card transactions and other business data services. This failure lasted 26 hours. Again, the bug was introduced during a software upgrade.

Description of the Problem

The code snippet that caused the outage is illustrated as follows

```
1. do {
2.     . . .
3.     switch (expression) {
4.         case 0: {
5.             if (some_condition) {
```

```
6.                  . . .
7.                  break;
8.              } else {
9.                  . . .
10.             }
11.           . . .
12.           break;
13.       }
14.     . . .
15.   }
16.   . . .
17. } while (some_other_condition);
```

In this case the break statement at line 7 was the culprit. As implemented, if the logical_test on line 5 was successful, the program should have proceeded to line 6 to execute those statements. When the program stepped to line 7, the break statement caused the program to exit the "switch" block between line 3 and line 15, and proceeded to execute the codes in line 16. However, this path of execution was not the intention of the programmer. The programmer intended the break statement in line 7 to break the if-then clause; so, after the program executed line 7, it was supposed to continue execution in line 11.

AT&T statement about the Problem

"We believe that the software design, development, and testing processes we use are based on solid, quality foundations. All future releases of software will continue to be religiously tested. We will use the experience we've gained through this problem to further improve our procedures."

We do not believe we can fault AT&T's software development process for the 1990 outage, and we have no reason to believe that AT&T did not rigorously test its software update. In hindsight, it is easy to say that if the developers had only tested the software, they would have seen the bug. Or that if they had performed a code inspection, they would have found the defect. Code inspection might have helped in this case. However, the only way the code inspection could have uncovered this bug is if another engineer saw this particular line of code and asked the original programmer if that was his or her intension. The only reason that the code reviewers might have asked this question is if they were familiar with the specifications for this particular code block.

12.5Art and Science of Debugging

Debugging is taken as an art but in fact it is a scientific process. As people learn about different defect types and come across situations in which they have to debug the code, they develop certain heuristics. Next time they come across a similar situation, they apply those heuristics and solve the problem in lesser time and with a lesser effort. While discussing the debugging process we discuss the phenomenon of "you miss the obvious". When a person writes a code, he develops certain impression about that code. One can term this impression as a personal bias that the developer builds towards his creation the

"code" and when he has to check this code, he can potentially miss out obvious mistakes due to this impression or bias. Therefore, it is strongly recommended that in order to reach to a defect in the code, one needs "another pair of eyes". That is, start discovering the defect by applying your own heuristics and if you could reach to the problem, fine, otherwise ask a companion to help you in this process. We shall further elaborate this idea based on the following example.

Program at Bulletin Board Example

Following piece of code was once placed on a bulletin board making an invitation to people to discover the problem in this code. Please look at this code and discover the problem

```
1. while (i =0; i < 10; i++) {
2.     cout << i << newl;
3. }
```

Well if you could not guess it by now, the problem lies with the syntax of the while loop. This is so obvious that almost everyone forgot to ponder upon when placed on the bulletin board of a university. The loop is "while" but the syntax used is a "for" loop.

In order to reach such defects, one needs a scientific approach to check and verify the code methodically. Based on this discussion, we are now in a position to introduce a few classes of bugs to the reader. This is not an exhaustive list since there could be a number of other classes of bugs as well but the following classes will help the reader know about some well known bugs that we usually find in the code.

Lecture No. 43

12.6 Bug Classes

Memory and resource leak

A memory leak bug is one in which memory is somehow allocated from either the operating system or an internal memory "pool", but never deallocated when the memory is finished being used.

Symptoms

- System slowdowns
- Crashes that occur "randomly" over a long period of time

Example 1

Let's take a look at a simple memory leak error that can occur trivially in a C program. This type of code is found in hundreds of programs across the programming spectrum. It illustrates the simplest possible case of a memory leak, which is when memory is allocated and not deallocated in all cases. This particular form of the bug is the most frustrating because the memory is usually deallocated, but not always

```
char *buffer = new char[kMaxBufferSize+1];
memset(buffer,0,kMaxBufferSize+1);

// Do some stuff to fill and work with the character buffer

if (IsError(nCondition)) // Did we get an error in the processing piece?
{
  Message(("An error occured.
            Skipping final stage"));
  return FailureCode;
}

// Final stage of code

// Free up all allocated memory

delete buffer;
return okCode;
```

Note that in many cases, this code works perfectly. If no error occurs in the processing stage (which is the norm) the memory is freed up properly. If, however, the processing stage encounters an error, the memory is not freed up and a leak occurs.

Example 2

The following code snippet from some old C++ code contains a slightly more insidious bug. It illustrates the point well.

```
Class Foo
{
  private:
    int nStringLength;
    char *sString;
  public:
    Foo()
    {
      nStringLength = 20; //Default
      sString = new
      char[nStringLength + 1];
    }
    ~Foo()
    {
      delete sString;
    }
    void SetString(const char
                   *inString)
    {
```

```
        sString = new char
                  [strlen(inString+1)];
        if(inString == NULL)
          return;
        strncpy(sString, inString,
                strlen(inString));
        nStringLength =
               strlen(inString)+1;
    }
};
```

Let's discuss what happens. In most cases, the Foo object is created and nothing bad happens. If, however, you call the SetString method, all bets are off. The previously allocated string is overwritten by the new allocation, and the old allocated memory goes nowhere. We have an instant memory leak. Worse, we can do this multiple times if we accidentally call the method with a NULL string because it first allocates the block, and then checks to see if the input string was correct. Don't do things like this. If you see an allocation in a class, check to see if the string can be allocated before it gets to that point.

Logical Errors

A logical error occurs when the code is syntactically correct but does not do what you expect it to do.

Symptoms

- The code is misbehaving in a way that isn't easily explained.
- The program doesn't crash, but the flow of the program takes odd branches through the code.
- Results are the opposite of what is expected.
- Output looks strange, but has no obvious symptoms of corruption.

Example

```
// Make sure that the input is valid. For this value, valid ranges are 1-10 and 15-20

if((input >= 1 && input <= 10) &&
  (input >= 15 && input <= 20))
  {
    // Do something for valid case
  }
else
  {
    // Do something for invalid case
  }
```

In order for the code to enter the valid case, the number must be between 1 and 10 and be between 15 and 20. If a number is between 1 and 10, how can it possibly be between 15 and 20 as well? Seems unlikely, doesn't it? This is a typical logical error.

Coding errors

A coding error is a simple problem in writing the code. IT can be a failure to check error returns, a failure to check for certain valid conditions, or a failure to take into account other parts of the system. Yet another form of a coding error is incorrect parameter passing and invalid return type coercion.

Symptoms

- Unexpected errors in black box testing.
- The errors that unexpectedly occur are usually caused by coding errors.
- Compiler warnings.
- Coding errors are usually caused by lack of attention to details.

Example

In the following example, a function accepts an input integer and converts it into a string that contains that integer in its word representation.

```
void convertToString(int InInteger,
      char* OutString, int* OutLength)
{
  switch(InInteger){
     case 1: OutString = "One";OutLength = 3;
               break;
     case 2: OutString = "Two";OutLength = 3;
               break;
     case 3: OutString = "Three";OutLength = 5;
               break;
     case 4: OutString = "Four";OutLength = 4;
               break;
     case 5: OutString = "Five";OutLength = 4;
               break;
     case 6: OutString = "Six";OutLength = 3;
               break;
     case 7: OutString = "Seven";OutLength = 5;
               break;
     case 8: OutString = "Eight";OutLength = 5;
               break;
     case 9: OutString = "Nine";OutLength = 4;
               break;

     }
}
```

There are a few things to notice in the preceding code. The ConvertToString function does not handle all cases of inputs, which becomes very obvious you pass a zero value. Worse, the function does not initialize the output variables, leaving them at whatever they happened to be when they came into the function. This isn't a problem for the output string, necessarily, but it will become a serious issue for the output length.

Memory over-runs

a memory overrun occurs when you use memory that does not belong to you. This can be caused by overstepping an array boundary or by copying a string that is too big for the block of memory it is defined to hold. Memory overruns were once extremely common in the programming world because of the inability to tell what the actual size of something really was.

Symptoms

- Program crashes quite regularly after a given routine is called, that routine should be examined for a possible overrun condition.
- If the routine in question does not appear to have any such problem the most likely cause is that another routine, called in the prior sequence, has already trashed variables or memory blocks.
- Checking the trace log of the called routines leading up to one with the problem will often show up the error.

Example

This particular example shows not only a memory overrun, but also how most programmers "fix" problems in an application. The ZeroArray function steps all over the array boundaries by initializing 100 separate entries. The problem is that that particular array only has 50 slots available in its allocation. What happens at that point is that the function goes past the end of the array and starts to walk on things beyond its control.

```
const kMaxEntries = 50;
int gArray[kMaxEntries];
char szDummyBuffer[256];
int nState = 10;

int ZeroArray (int *pArray)
{
  for (inti=0;i<100;++i)
    pArray[i] = 0;
}
```

Loop Errors

- Loop errors break down into several different subtypes.
- They occur around a loop construct in a program.
- Infinite loops, off-by-one loops, and improperly exited loops.

Symptoms

- If your program simply locks up, repeatedly displays the same data over and over, or infinitely displays the same message box, you should immediately suspect an infinite loop error.
- Off-by-one loop errors are quite often seen in processes that perform calculations.
- If a hand calculation shows that the total or final sum is incorrect by the last data point, you can quickly surmise that an off-by-one loop error is to blame.
- Likewise, if you were using graphics software and saw all of the points on the screen, but the last two were unconnected, you would suspect an off-by-one error.

- Watching for a process that terminates unexpectedly when it should have continued.

Example

```
bool doneFlag;
doneFlag = false;
while(!doneFlag){
  ...
  if( impossibleCondition )
    doneFlag = true;
}
```

The preceding code fragment contains an indirect looping error such that the loop will continue until the impossibleCondition becomes true which is impossible to happen.

```
int anArray[50];
int i;
i = 50;
while(i >= 0){
  anArray[i] = 0;
  i = i - 1;
}
```

In this example, the programmer was trying to be smart. He worked his way backward though the array, assuming that this way there would be no chance of an error. Of course, if you examine the loop, you will find that it performed 51 times. There are only fifty elements in the array. This will certainly lead to a problem later on in the program, if it doesn't first trigger an immediate error from the system.

```
int nIndex = 0;
for(int i=0; i<kMaxIterations; ++i)
{
  while (nIndex < 20)
  {
    ComputeSomething(i*20 + nIndex);
    nIndex ++;
  }
}
```

The final situation is an improper exit condition for a loop. This is most easily illustrated using a set of nested loops. In the above example, we are trying to compute something within the inner loop for some number of iterations. The code appears to do what we said it should do. It computes whatever it is we are computing in the inner loop for each iteration of the outer loop, 20 times. The problem, however, is that the exit condition simply says that nIndex should be less than 20. The first time through the outer loop, this variable will become 21, and the inner loop will exit. This is correct, and exactly the way

you would expect this to happen. The problem, however, is that the next time through the loop, the inner loop will not be executed at all.

Pointer errors

- A pointer error is any case where something is being used as an indirect pointer to another item.
- Uninitialized pointers: These are pointers that are used to point at something, but we fail to ever assign them something to point at.
- Deleted pointers, which continue to be used.
- An Invalid pointer is something that is pointing to a valid block of memory, but that memory does not contain the data you expect it to.

Symptoms

- The program usually crashes or behaves in an unpredictable and baffling way.
- You will generally observe stack corruptions, failure to allocate memory, and odd changing of variable values.
- Changing a single line of code can change where the problem occurs.
- If the problem "goes away" when you place a print statement or new variable into the code that you suspect contains the problem.

Example

Let's take a look at one of the most common forms of the pointer error, using a pointer after it has been deleted. As you can see by the following example, it is not always clear when you are doing this incorrectly:

```
void cleanup_function(char *ptr)
{
  SaveToDisk(ptr);
  delete ptr;
}

int func()
{
  char *s = new char[80];
  cleanup_function(s);
  delete s;
}
```

In this example, the programmer meant to be neat and tidy. He probably wrote the code originally to allocate the pointer at the top of this function, and then by habit put in a deallocation at the bottom of the function. This is, after all, good programming practice to avoid memory leaks. The problem is introduced with th3e cleanup_function routine. This function was probably written at a later time by another developer and might have been used to ensure that all allocated pointers were stored to the permanent drive and then freed up to avoid any possible leaks. In fact, the code in question comes from a set of functions that was reused from a memory pool system that saved its data to disk before destroying the pointer, so that another pointer could be retrieved from the persistent pool.

The problem is, the reuse did not take into account the existing system. This problem is common when reusing only parts of a system.
When the second delete occurs at the bottom of func(), the results are unpredictable. At best, the delete function recognizes that this pointer has been deleted already and doesn't do anything. At worst, memory is corrupted and you are left with a memory crash somewhat later in the program, which will be very difficult to find and fix.

Boolean bugs

Boolean bugs occur because the mathematical precision of Boolean algebra has virtually nothing to do with equivalent English words.
When we say "and", we really mean the boolean "or" and vice versa.

Symptoms

- When the program does exactly the opposite of what you expect it to. For example, you might have thought you needed to select only one entry from a list in order to proceed. Instead, the program will not continue until you select more than one. Worse, it keeps telling you to select only one value.
- For true/false problems, you will usually see some sort of debug output indicating an error in a function, only to see the calling function proceed as though the problem had not occurred.

Examples

The following code contains a simple example of a function that returns a counter-intuitive Boolean value and shows how that value leads to problems in the rest of the application code. This case performs an action and indicates to the user whether or not the action succeeded.

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

After looking at the function, ask yourself this – What is the return value in the case of success? What would it be in the case of failure? How would you know this by simply glancing at the function declaration?

```
int Value;
int Ret;

Value = 11;
```

```
Ret = DoSomeAction(Value);
if(Ret)
  cout <<Error in DoSomeAction<<;
```

If you were debugging this application, you might start out by wondering why it didn't work. Further investigation would show that the error statement is never triggered in spite of the fact that the routine did, in fact, encounter an error.. Why did this happen? It happened because the routine didn't return nonzero value for failure, it returned nonzero for success. This Boolean value was not intuitive given the condition of the function and led to a poor assumption by the programmer who wrote the code that used it.

Lecture No. 44

12.7 Holistic approach

Holistic means

- Emphasizing the importance of the whole and the interdependence of its parts.
- Concerned with wholes rather than analysis or separation into parts

What this meant is that a holistic approach focuses on the entire system rather than whatever piece appears to be broken. Holistic medicine, for example, concerns itself with the state of the body as a whole, not the disease that is currently attacking it. Similarly, programmers and debuggers must understand that you cannot treat the symptoms of a problem, you must focus on the application system as a whole.

Now that you understand what holistic means, how do you apply holistic concepts to debugging the software programs and procedures? Let's start by thinking about what emphasizing the whole over the individual pieces means in debugging. When you are debugging an application, you often simply look at the problem reported by the user and how you can make that problem go away.

As a simple example, consider the case where the program mysteriously crashes at a particular point in the code each and every time the program is run. The point at which the code crashes makes no sense at all. For example, in C++, you might have a member function of a class that reads:

```
void SetX(int X)
{
  mX = X;
}
```

The program always crashes on the line that assigns X to the member variable mX(mX = X). Looking at this code snippet, we see very few valid ways in which the program could be encountering a problem at this point. Oddly, we might discover that inserting a message statement makes the problem stop occurring when the indicated steps are followed for reproducing the problem. Is the problem fixed?

Most experienced programmers, managers, or debuggers would say that the problem is not fixed, although few could tell you exactly why that is the case. The fact is, in this case, we are directly treating the symptoms (the program crashing) rather than looking into the actual problem and ignoring the overall problem is endemic in our industry.

12.8 The debugging process

In normal circumstances, you will have a user description of the problem. This description might have been given to you directly by the user, or it might have been gathered by a customer support person or other no technical person, In any event, this data has to be considered suspiciously until you can get a first hand description of what really happened. First hand accounts of the problem are always useful, so be sure to write down exactly what you are told. That way, you can compare several accounts of the same problem and look for similarities. Consider, for example, the following accounts of a reported bug in a system:

1. "I started by trying to setup a Favorites list. I first went to the home page. Then, I selected Favorites from the menu on the right. I scrolled down to the third entry and pressed Enter on the keyboard. Then I moved to the fourth entry on the submenu and clicked it with the mouse. Finally, I entered my name as Irving, clicked on OK, and the program crashed."

2. "I went to the programming menu and selected the New menu option. I then clicked on the Create Object menu entry and entered the name HouseObject for the object name field. Then I clicked OK, and the program crashed."

3. "I selected the New Project menu option, and then clicked on the icon that looks like a Gear. I entered the name Rudolph for the project name, and then clicked OK. The program crashed."

    - After all, they appear to relate to three different sections of the program.
      Most people, when reporting bugs, focus on what they think was the important step in the process.
    - Filter out the unimportant information and see what each case really has in common.
      First, each user clicked on the OK button to finalize the process, and the program crashed.
    - It might seem likely, therefore, that the program OK handler contains a fatal flaw.
    - A bit of experimentation will show whether this is the case or not.
    - When we examine the statements of the witnesses, we notice that each was working with a menu.
    - We can see that each person used both the keyboard and the mouse to select from the menus before clicking the OK button.

"I selected Favorites from the menu on the right. I scrolled down to the third entry and pressed Enter on the keyboard. Then I moved to the fourth entry on the submenu and

clicked it with the mouse."

"... selected the New menu option. I then clicked on the Create Object menu entry ..."

"I selected the New Project menu option, and then clicked on the icon that looks ..."

- We can infer, therefore, that the user in this case used both the keyboard and mouse.
  The next logical place to look is in the menu handler to see whether it deals with mouse and keyboard entries differently.
- The final clue is in the third entry, where the user did not select anything from the menu with the mouse, but instead clicked on an icon.
- While looking at the code, I would try to see what happens when the program deals with a combination of keyboard and menu entry.
- It is likely, given the user discussion, that you will find the root of the problem in this area.
- After finding such a common bug, it is likely that the fix will repair a whole lot of problems at once
- This is a debugger's dream.

Good clues, Easy Bugs

Get A Stack Trace

In the debugging process a stack trace is a very useful tool.
Following stack trace information may help in debugging process.

- Source line numbers in stack trace is the single, most useful piece of debugging information.
- After that, values of arguments are important
  - Are the values improbable (zero, very large, negative, character strings with non-alphabetic characters?
- Debuggers can be used to display values of local or global variables.
  - These give additional information about what went wrong.

Non-reproducible bugs

- Bugs that won't "stand still" (almost random) are the most difficult to deal with.
- Randomness itself, however, is information.
- Are all variables initialized? (random data in variables could affect output).
- Does bug disappeared when debugging code is inserted? Memory allocation (malloc) problems are probably a culprit.
- Is the crash site far away from anything that could be wrong?
- Check for dangling pointers.

Example

```
char *msg(int n, char *s)
{
  char buf[100];
```

```
   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