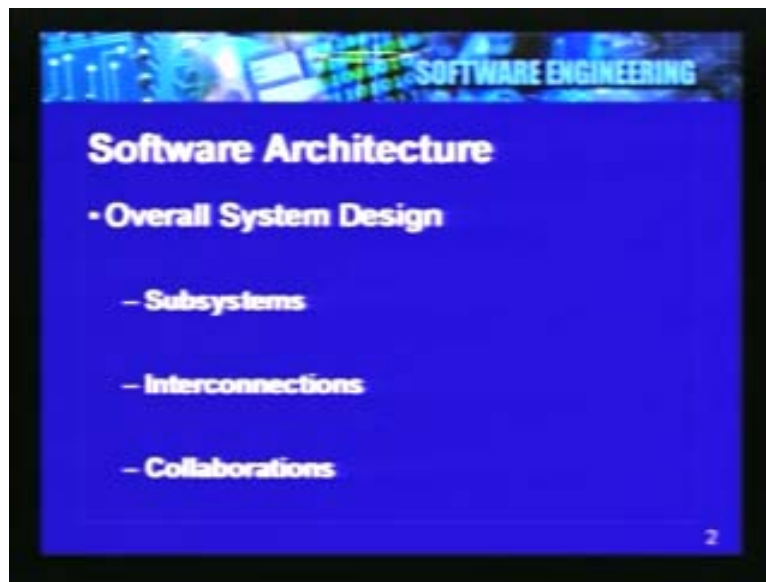


Software Engineering
Prof. Rushikesh K. Joshi
Computer Science & Engineering
Indian Institute of Technology, Bombay
Lecture - 17
Architectural Design

In this lecture we will talk about architectural design.

(Refer Slide Time: 01:02)



What is architecture? Architecture gives us overall system design. For example, before you actually design the internal subsystems of given software, before you go on to the detailing, you want to grasp the overall design of the software. So the architecture is a very important aspect of software design. We can talk about subsystems, interconnections among subsystems and collaborations among sub systems.

You must have seen architectural diagrams. Architecture is a very common technique to understand a complex system and then to approach system building from architectural point of view, that is top down. Architecture as a concept is applied in many disciplines. For example, you see building architecture before the building is actually built. And after the building is actually built, then the architecture gets refined over time and then you get all the diagrams of the actual construction. Why do we need architecture? We are summarizing the need for architecture in the below slide.

(Refer Slide Time: 02:30)



Firstly, in order to understand a given system. Systems are very complex and they may have thousands of components, classes. So it is very difficult to grasp such a system from bottom up view, say right from low level functions or data structures. If you take for example, an operating system source code, how will you go about understanding such a source code? Will you start reading directly a specific module and then build your understanding about how the entire system is organized or will it be easier to understand system if you had an architectural document with you? How many modules are there in the system? How are they connected? What is overall perspective on your chosen software?

So an architectural document or architecture of the system makes it possible to understand a complex system, you can organize your development. So understanding is very important if after the software is built and if changes are to be required. And before the software is actually built, you want to organize the development itself according to architectural partitioning. For example, in architecture you have identified a few servers, client side, components, middleware, and then you can partition your development accordingly.

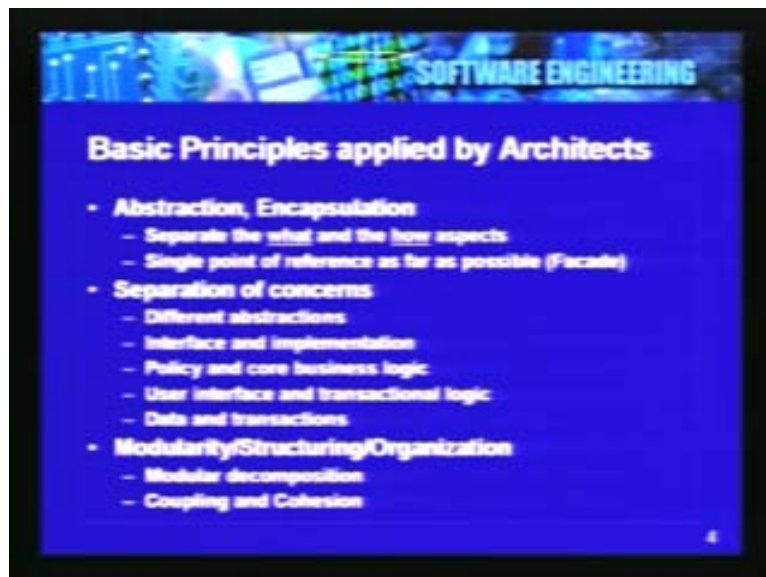
You can organize your programmers according to architectural partitioning. It becomes easier to handle the system if architecture is available or if architectural approach is taken at the beginning. And then it also promotes reuse by componentization. In architecture, if you identify a few components which are already built, then you benefit from their reuse. When you are architecting a system, you have to also take into account similar products which have been written earlier in your organization or which are available through third parties or they could be say class libraries and packages and so on.

When you draw your architectural diagrams, when you do your architectural analysis, at that time you can benefit from what is existing and you have to also do an analysis of reuse. Those reusable components, packages or whatever entities you are reusing in your architecture depends on what level of architectural you are handling. If it is highest level of architecture, there could be highest level of architectural components which are reusable. So at that level one has to identify what you are going to reuse and what you are going to build. Hence architecture is very important from reuse perspective.

And again, when you are architecting a system you might also take into account future reuse in a given product line. A product improves over time because requirements get refined and refined over versions. Then bugs get found out, and then you might want to reengineer some parts of the system and so on. A product always improves in terms of functionality, performance, its correctness and so on. So architecture becomes very important it plays a major role in a product line. Evolution is another important aspect that benefits from architecture which is changes and dependencies in the system. The Overall dependencies among the modules are identified in the architecture or the dependencies among the components that have been identified in the architecture.

The architecture also draws the evolution process. Product can evolve due to versioning or it could be due to specialization and so on. We are going to subsequently discuss different kinds of architectural patterns. Some of these ideas will come again when we look at some specific example of patterns of architecture. What are the basic principles which are applied by architects?

(Refer Slide Time: 07:27)



These are some of the very important principles which need to be taken into account when you are architecting the system.

Abstraction and encapsulation: You must be able to identify the right abstraction at architectural levels and then hide the internal information or the internal details through encapsulation. This is about separating the 'what' and the 'how' aspect; what does the component do and how it does. If you identify a specific package say the GY package, you should know what it does and then separate how the actual business logic, the actual logic the GY components are implemented. For the GY package, you will be defining the classes which are available from that package. Then there may be internal classes in order to implement the projected behavior outside the given package. You have to separate what module does from, how it is done.

Then single point of reference as far as possible. When you have a sub system and there is a single point of reference into the sub system, it becomes the reference easier to manage dependencies. If something changes inside the subsystem it could be reflected through only the single point of reference. If from outside and if there are many references into a subsystem and if the sub system undergoes changes there will be many dependencies directly outside through multiple points of contact. This is another important aspect of abstraction and encapsulation. Then a general principle of separation of concerns has to be applied when you are architecting the system.

You must separate different concerns. For example different abstractions identify different constructs. Every module or every library or every package does different things and the responsibilities have to be separated out clearly. Separation of interface from implementation in order for the external environment or the calling environment only looks at interface and identifies the component through its interface. Implementation is different and is separated. If you separate interface from implementation you get various benefits. For example implementation can change and still the calling environment sees no change because interface is the same.

Then separation of concern can be applied for say policies and for the core business logic of the component or the system. The policies are different and the core logic is different. You have to list out and identify policies separately and you should have a separate architecture for policy implementation. For example, security policies such as your login, password, user accesses like, who can access what, the access matrix and so on. So if you have policy architecture separate from the core business logic or from the core business component architecture, then you are separating the two concerns and it becomes easier to grasp such a system. You have to identify different concerns and give different architectures for these and say what the dependencies amongst them are.

The separation of concerns can also be applied to user interfaces and transactional logic. You separate interfaces from transactional logic or the control logic from the interface the system is used. The interface architecture is different from the actual transactional logic. From the interface itself you may not directly operate on the low level data entities, but you may delegate calls on to the transaction logic and the transactions. Then implement the operations, the functions which implement this business logic. They take care of requirements of transactions, they identify the data and then operate on the data they take care of concurrency and so on.

So user interfaces are separate from transaction logic. Similarly transactions and data again are separate. The data modeling aspect, the entity relationships can be a separate and transactional dependency, which is a process level architecture can be given separately. Again, if you separate the concerns it becomes easier to grasp such a system and then according to the separation, you can also manage your development. So this is a very important principle and an experience architect finds out what are the different concerns at from different points of use. For example, abstractions, interfaces and implementation, policies, core logic or transaction logic, user interfaces, data modeling and so on.

These are various concerns. Another important principle that is to be applied by architects is modularity; structuring an organization. This is much related to abstractions. When you find your abstractions, you are modularizing and then you structure them. You structure these abstractions, you structure the module. Module is typically very close to an implementation level concept. You can give a module as, for example a file implementing a few functions and then you have structure amongst the modules and that is your organization of the modular system.

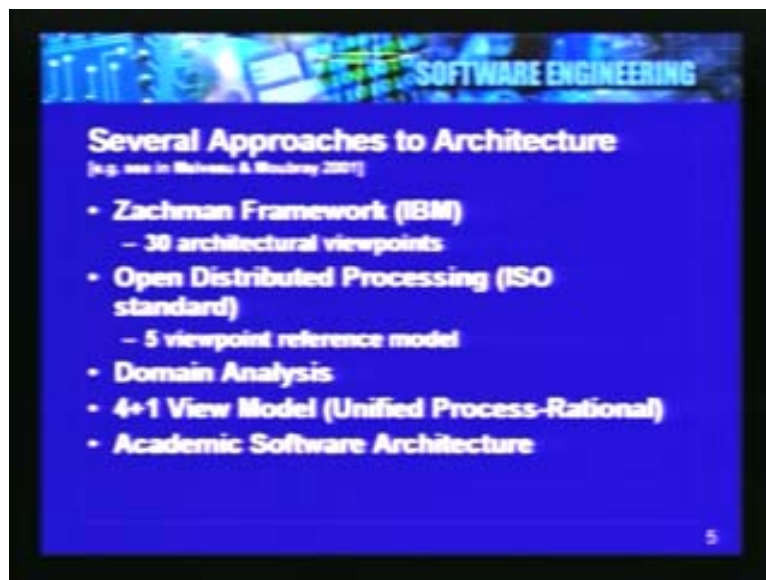
So you can look at modularity/structuring/organization and these are different aspects of structuring. Independent modules which are structured give you an organization of modules. The decomposition has to be modular. Again you are applying separation of concerns, but now you are talking about slightly more physical architecture or physical systems in terms of modules. Modules are coupled together. So how far should they be coupled? Should two modules depend on each other too much? And within a module, for example if you are looking at a class as a module, in the class there are private members and there are public member functions.

Public member functions access your implementation, that is your private data and it may be possible that you can partition your public member functions into two partitions such that they are independent and they access totally two different sets of private data structures. In this case the module is not cohesive. It is doing two different things. So you have not really separated the concerns. A module which is a physical manifestation, gives an abstraction. It is not really a single abstraction, it may be a composition of two independent abstractions, but may have been put together for the sake of convenience.

But such modules might inject a few problems into the system's architecture because when one role wants to access only one specific activity or specific interface on that module, it also gets access to some thing else which is not required by that role, then your breaking encapsulation of the system because you are exposing more than what is required for a given role. So it is very important to design your modules, keep them cohesive. The roles which access these modules play a major part in this module design. For example in object oriented design you could use multiple inheritance and then if you have multiple inheritances of interfaces if you have two three interfaces and one implementation of this class. That implementation shares common state or common data structures in order to implement all the three interfaces and again the module is still cohesive.

Then the coupling amongst different modules that is interdependencies amongst the modules must be as low as possible. If the modules depend too much on each other and if there is too much coupling, then changing one independently without affecting other module, without having to change the other module becomes difficult or impossible. So coupling and cohesion are very important aspects in modularity. Several approaches have been proposed to architecting systems and various books have been written on this. One such reference has been mentioned on this slide and you could go through the list of the references which I have used during this presentation and the list has been given at the end of this lecture.

(Refer Slide Time: 17:12)



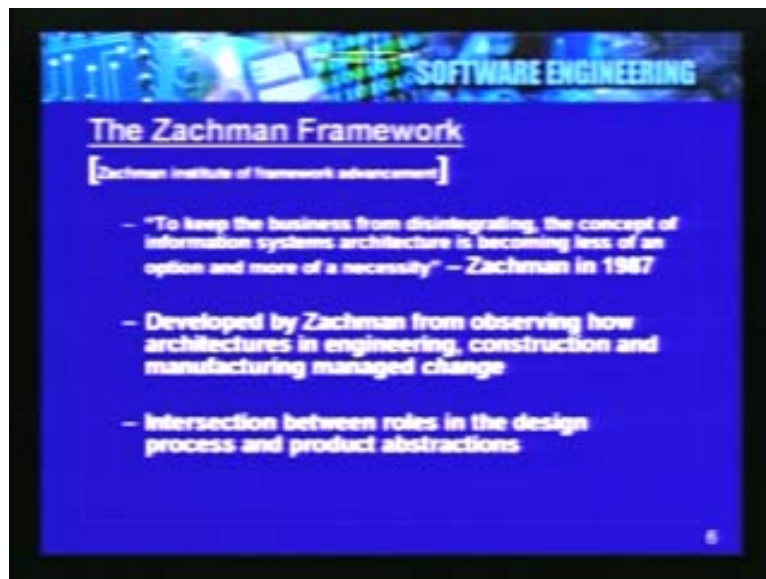
We will briefly look at some of these approaches to architecture. Zachman's framework is a very important approach to architecture. How to go about architecting a system, what needs to be architected, how many view points are there for a given system, is there only one architecture or there are many view points? For example when you model your software, you have to do static modeling, dynamic modeling, you draw class diagrams, object diagrams, interaction diagrams, activity diagrams, flowcharts, DFD's, entity relationships, state diagrams and deployment diagrams.

All these models give you different views of the same software. How many such views do you have for architecture? Is there one unique view for software or are there many views? What are the different aspects which have to consider when you are architecting a system? Many approaches have been proposed and this area is still evolving. And there is lot of academic discussion on what is to be architected and how it is to be architected, what are the basic primitives, what are the basic connectors, what are the architectural description languages, how to model or how to represent your architectural models pictorially? There is lot of research going on in academic community on architecture. We are going to look at some of the existing approaches to architecture.

One important approach is Zachman's framework. It gives us 30 architectural view points. From thirty different view points, you could get 30 different architecture for the given system. We will look at what are those later. Then there is ISO standard called 'Open Distributed Processing'. This gives an approach to architecting distributed systems. It gives you mainly 5 view point reference model and then there is domain analysis approach, when you mainly look at the domain. For example, there are agent oriented systems where you do an analysis based on agents and what are the entities in the domain and how agents are related to those entities.

For example if you are building a library information system, you will firstly look at who are the actual agents, which are the human beings who operate or who are involved in such a system. So without having to start with the software directly, you first model the domain itself and do analysis of that and then get the domain architecture. And then you can convert it into software architecture. That is a different approach. Then you have a 4 plus 1 view model, which is what was adopted in the unified process architectural model. There is lot of academic software, architecture research, which gives us many different approaches, architectural description language and so on. We will have a look at some of these.

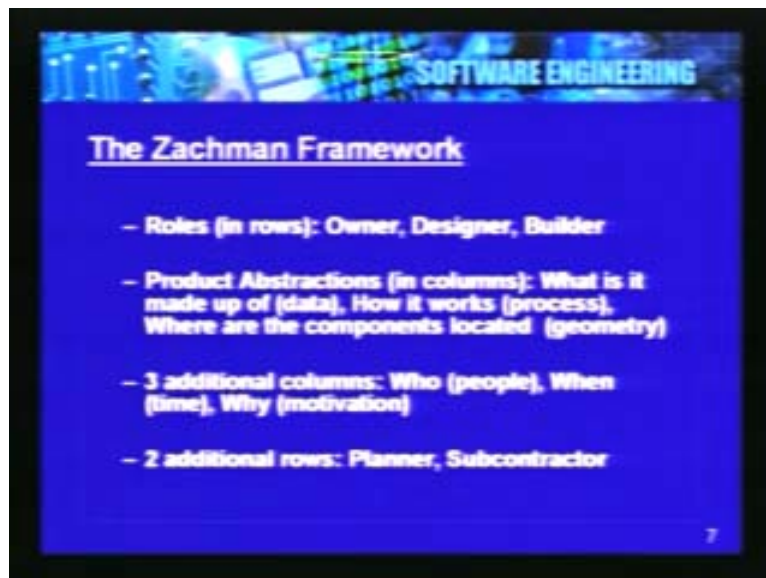
(Refer Slide Time: 21:14)



Now the Zachman's framework, you can go through the website of Zachman's institute of framework advancement. There is a poster available there on Zachman's framework. I will show you the poster quickly. Before that we will just see what the motivation for this framework was. Zachman say in around the year 1987, noted that to keep the business from disintegrating the concept of information systems architecture is becoming less of an option and more of a necessity. In order to keep your information together, in order to keep your business together, you need architecture; you need to organize everything very well. Otherwise it will disintegrate soon. If you organize, you benefit from the organization. So architecture was seen as a necessity.

This was a very important quote by Zachman and he developed this Zachman's framework by observing how architecture engineering construction and manufacturing managed changes. He mainly defined 30 view points based on roles and product abstractions. There is a role cross product abstraction matrix and roles are represented in rows.

(Refer Slide Time: 22:53)



For example, owner's role needs architecture from owner's point of view. Designer's role needs architecture from designer's point of view. Then builders were the one who actually build the system would see the system's architecture from builder's point of view. The builder is not interested in the software architecture which is from the owner's point of view. Then product abstractions are in columns. So mainly say what the product is made up of, the data, how it works, the process and where are the components located, the geometry, the placements and then there are three additional columns: Who- that is the people involved; When - which is about the time; And why - which is motivation and there are two additional roles, which are planners and sub contractors role.

If you take a role and product abstraction, you get a specific architecture and that architecture is useful for that role and for the given product abstraction or for given aspect of the product. Say if you are interested in builders- What, then you are looking at the actual entity relationships of data, so when you are talking about 'what', you are talking about data. The builder is looking at the data which he actually wants to build. So he may be looking at the data structures or the entity relations for designing of tables and so on. 'How' aspect captures the processes or the control part and the 'Where' aspect captures the locations. We will just quickly look at this Zachman's framework.

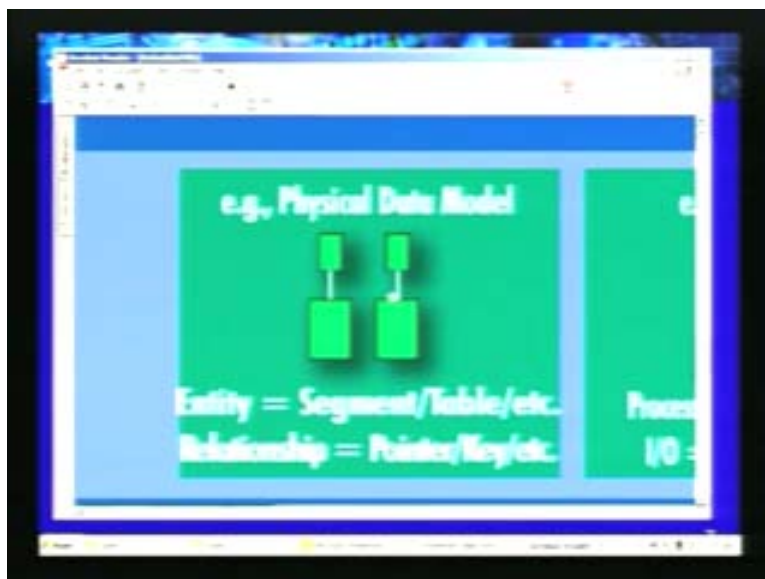
This is the poster from Zachman Institute of Framework Advancement and you can download this from www.zifa.com.

(Refer Slide Time: 25:12)



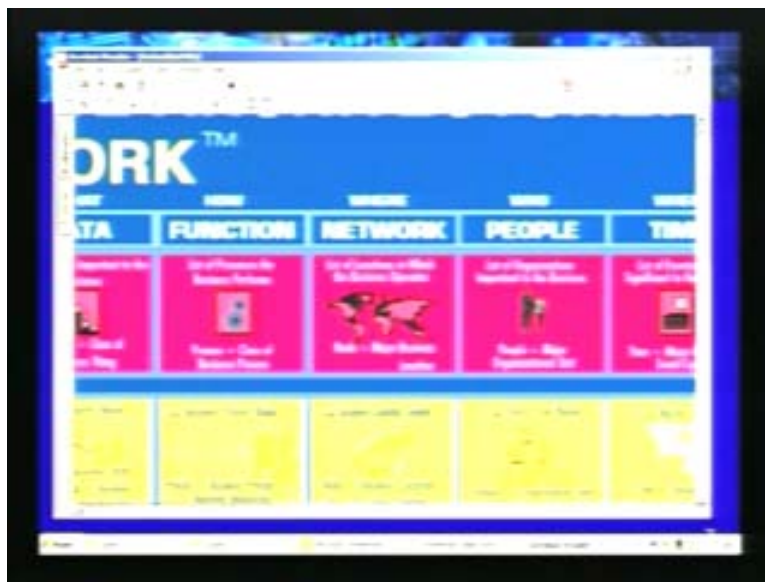
You can see those 30 different view points in this poster. Say we will look at builders' 'what' aspect. The builders' 'What' aspect is shown in green color (which is focused more closely in the below slide).

(Refer Slide Time: 25:59)



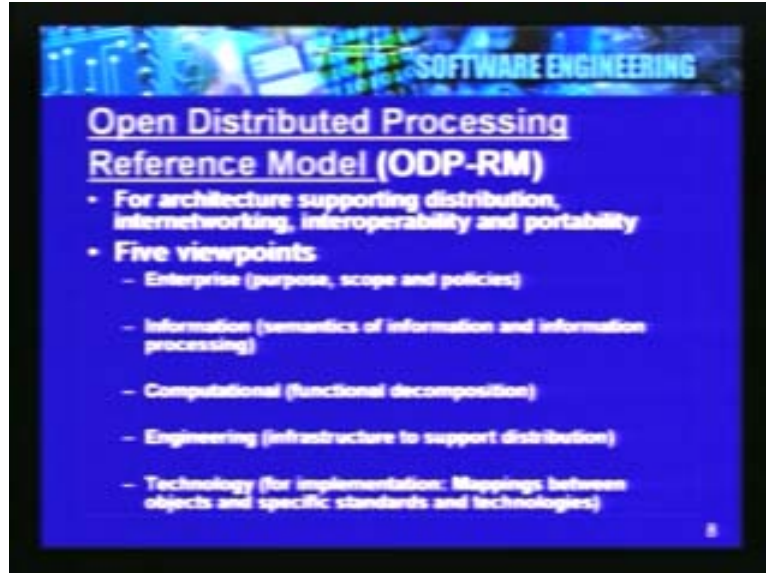
Looking at physical data model, you have the entities as the segments and tables and so on. Or if you have data structures, relationships will have pointers and if you have table, there are keys and so on. This is the builders' architecture. If you look at owners' architecture of 'what', you will get something different. Say for example, let us take a look at planners' and 'where' something interesting. If somebody is planning the software, owner is the one who owns the business and the planner is the one who plans about it. So the 'where' aspect you are looking at different business locations and if you go deeper, the 'where' aspect is going to give you more detailed or more focused places such as the actual machines or it could be even the actual components or even the addresses and so on.

(Refer Slide Time: 27:08)



What we can see in the Zachman's framework is that we have all these 30 different view points. We are not going into the details of all these viewpoints. You can get more information by reading the related slides and book. This was an important contribution to architecting software systems. This gives you very detailed and organized view about how to go about architecting systems. Now we will look at another related model or approach to architecting system. This Open Distribution Processing Reference Model is called as ODP-RM. This is for architecting system that supports distribution internetworking, interoperability and portability.

(Refer Slide Time: 28:02)



Typically, say you have distributed system connected in a LAN or in an enterprise which may be connected through an interoperable middleware and you have components distributed and then they access services of each other. The components may go down and come up in their lifecycle or some of the components may get deleted, the components may change the implementations, you may share the components in the system. If you look at an enterprise wide architecture, where there are machines distributed and services may be distributed and applications may be accessing these different services.

Then what are the approaches to architecting the systems? This is one approach to architecture. It provides you with the following 5 view points which are listed on this slide. Enterprise view point talks about the purpose of the software, its scope and different policies that are to be followed. The information view points talks about semantics of information and information processing. A computational view point talks about the functional decomposition, the decomposition of the system. We can say modular decomposition of the system.

Engineering view point talks about the infrastructure to support in the distribution. And the technology viewpoints talks about the mappings between objects and specific standards and technologies and so on. We will look at these view points one after the other. For example, distributed system architecture may give its enterprise view point which is directly understandable by managers and end users.

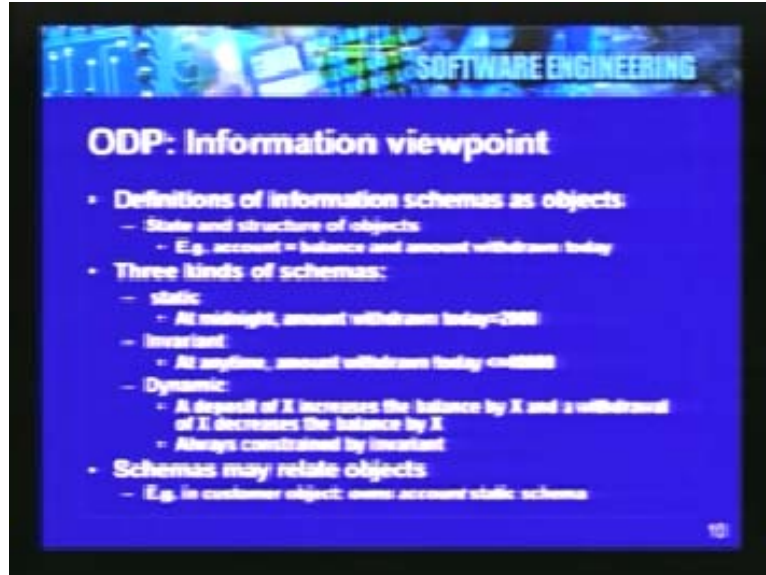
(Refer Slide Time: 30:16)



You are looking at the purpose of your business or your software which you are architecting its scope and policies. It includes different permissions, prohibitions and obligations. For example, you may have these active objects or human beings which are important in this business or for this given software which are managers, tellers and customers. The passive objects are accounts. Bank managers must advice customers when interest rate changes; this is an obligation. Cash less than 40000 can be drawn per day; this is prohibition. Money can be deposited; this is permission.

This is very high level view point and through this the mangers and the end users understand the system. This high level view point may be described in a description language and that gives you the architecture of the system or you may also draw pictorial representations of the same. Then information view point gives you definitions of information schemas as objects.

(Refer Slide Time: 31:47)

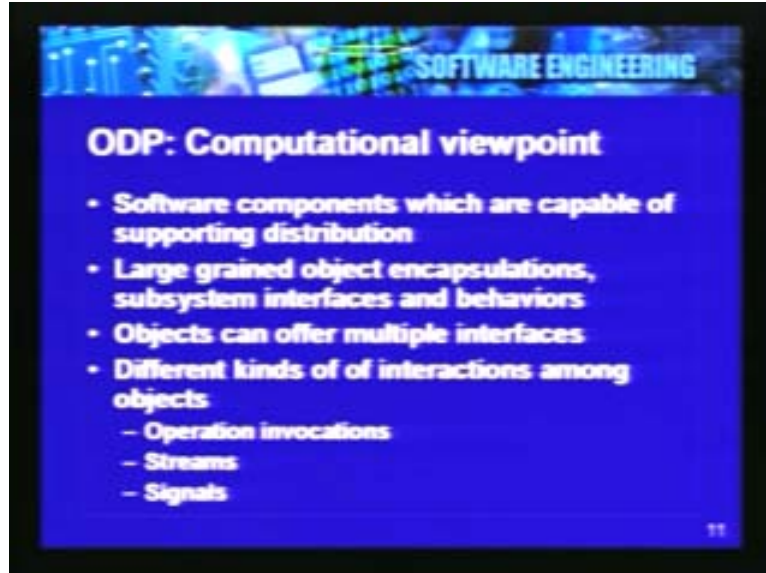


For example state and structure of objects. Say in the account object, you may have its state as balance, an amount withdrawn on that day or on any day. So amount withdrawn gives you the total amount that was withdrawn on the day of reference. Then there could be three kinds of schemas for this information view point. The static schema; for example at midnight amount withdrawn today is 2000. Invariance schema gives you an invariant for your given object. At any time amount withdrawn today should be less than or equal to 40000.

And the dynamic schema gives you the dynamics which are always constrained by the invariant. For example, if you deposit x then the balance increases by x. If you withdraw x, the balance decreases by x. So this is very similar to design by contracts where pre conditions, post conditions and invariants can be used. This is information viewpoint. When you are looking at informational objects or schemas, you are going to do an analysis from these three points of views; the invariants, the dynamics and the static attributes or **the static's** of your chosen objects.

And then the schemas may relate objects. For example in customer object, a customer object owns an account static schema. The dynamic schema might be governed by the policies of the bank. The customer object owns the account static schema, the actual balance variable, the actual state of the object. This is from information view point. Then from computational view point, you are talking about software components and how you are going to implement them.

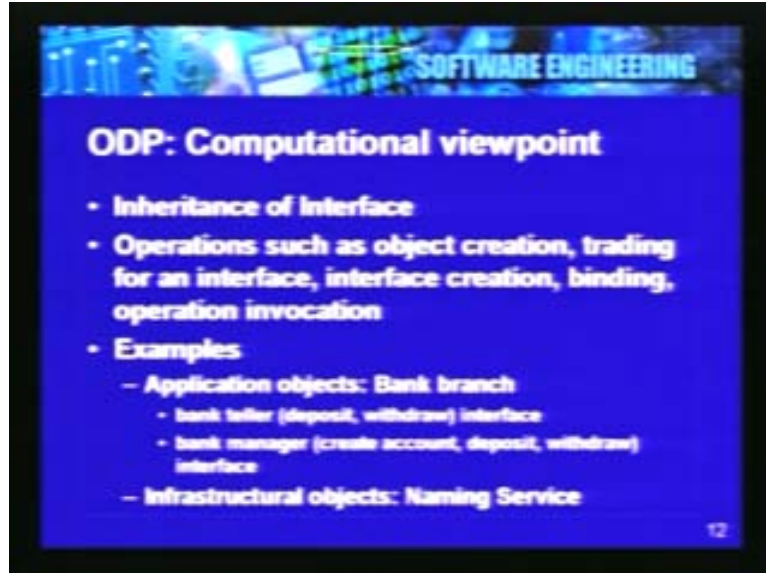
(Refer Slide Time: 34:06)



For example large grained object encapsulations; may be thought of as server processes or you could have sub system interfaces for packages. Then you can talk about how objects are implemented. How objects offer different multiple interfaces. How they are implemented to project various behaviors or various interfaces for different roles, how they interact with each other etc.

You are talking about implementation; how the calls are sent, how messages are sent from object to object or from module to module. Are they through message invocations or are they through exceptions or they continuous streams? So there are different kinds of connectors, you are talking about different components; how they have organized and how they are implemented, what are the relations among the interfaces and their implementations and how they are connected through different connectors. We are listing some more details about computational viewpoints below.

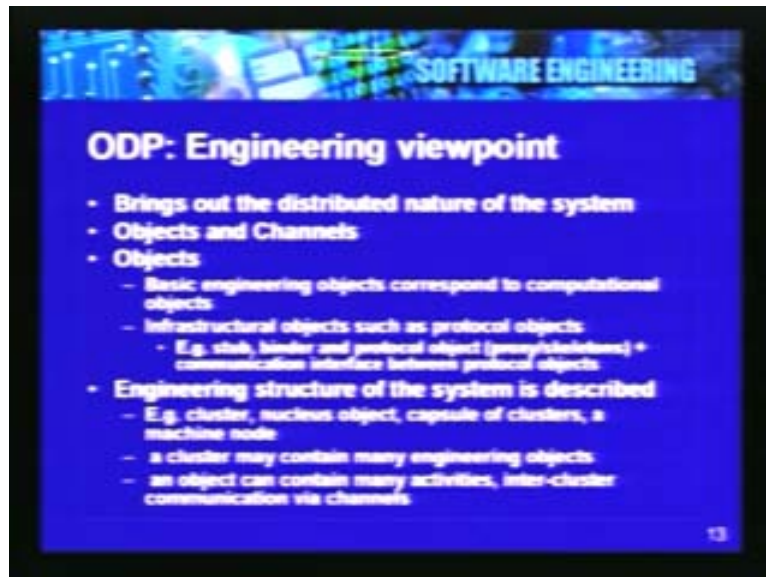
(Refer Slide Time: 35:47)



First one is the inheritance of interface. Then various aspects about objects such as how they are created, how do you create for interface, how do you search, how do you locate the object, how are interfaces created, how does the actual binding occur, how are operations invoked. So we are talking about a distributed system and then in the distributed system you have to actually detail out how the invocation goes from one machine to another. On application objects there could be different interfaces, say for example bank tellers' interface and bank managers' interface are different. Say you could allow deposit and withdraw on the teller, where as you can create an account through bank managers interface.

And there could be infrastructural objects involved from computational view point, from the actual implementation viewpoint. That is you are using some of the existing infrastructure services such as naming service event service and so on. Then the engineering view point brings out the distributed nature of the system. So when you are looking at computational view point, you are mainly looking at the objects and how are they implemented, how are they connected and what are the different computational connectors or relationships among them. When you are looking at engineering view point, you are actually looking at the physical manifestations in a distributed system.

(Refer Slide Time: 37:07)



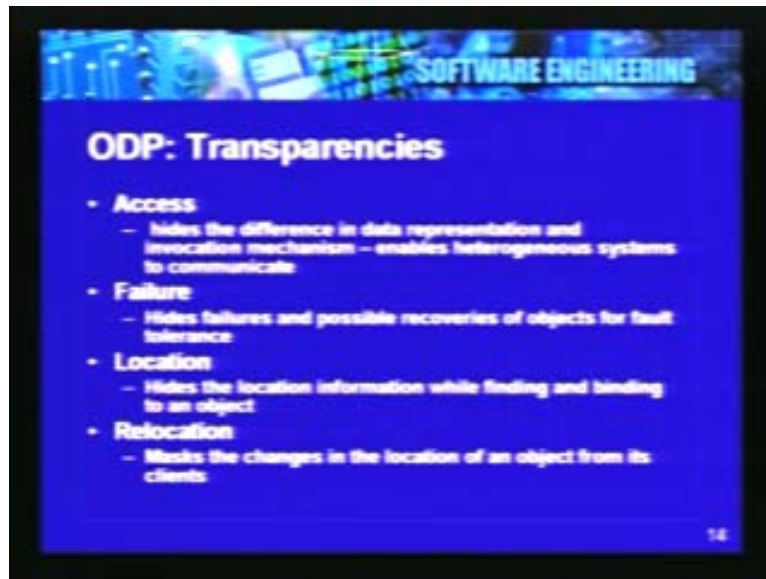
How they will be connected physically? What are the different objects from engineering view point: The objects, the channels and the actual connections? For example your objects may be basic engineering objects which are corresponding to the computational objects which are identified in computational viewpoint or infrastructural objects such as say protocol objects, your stubs, the proxies and skeleton or binders and communication interface between the protocol objects. The low level objects which are required to make it possible for a message to go from one system to another. This is the engineering view point. The objects in engineering view point come out when you are talking about the engineering view point architecture of the system and not in the computational view point.

The computational viewpoint mainly talks about that the functional requirements: What are the entities, what are the objects in the domain and the related processes and a relationship amongst those objects? But from engineering viewpoint you are actually making the system distributed and you are talking about, how this distributed system is going to function together, what are the actual objects, how are they located, where are they located? Say for example the engineering structure of the system could be identifying the clusters, the nucleus objects, the capsule of clusters or a machine or a cluster may contain many engineering objects or an object may contain many activities and then you have inter cluster communication via different channels. So you are talking about the physical aspects or the engineering aspects of the system.

You may have a server process in which many objects are held. But this does not come out in the computational viewpoint. There we are mainly talking about core computational objects and relations among them, whereas in engineering view point, we are talking about the engineering aspects of the system. So some more objects come out in this engineering view point.

So you have to carry over the objects from the computational view point and also add the engineering objects such as say proxies and skeleton which you do not have to detail when you are talking of the computational view point. If you get the entire architecture in one shot, then it will be too complex to comprehend. If you separate these viewpoints, you are looking at different kinds of architectures for different roles. So this is another approach to architecting and this is specifically for distributed systems. In this approach you have to also look into various transparencies guaranteed by your system.

(Refer Slide Time: 40:33)

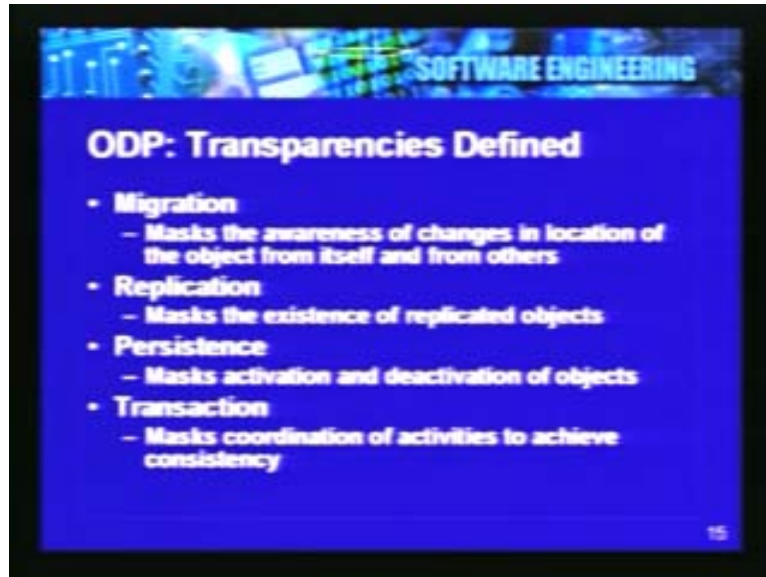


For example do you guarantee access transparency? Access transparency hides difference in data representation and invocation mechanism. For example, you may use your server through function calls, but your server may be object oriented which requires member function call or you may have an implementation which is in COBOL but you have your client which is in a object oriented programming languages, which understands classes, objects and invocation on objects. So this access transparency hides the actual data representation from invocation mechanism. The invocation mechanism is different from the actual data representation.

Then you may have failure transparency. When a component fails, the caller would not be aware of this failure. The component may come up. So that the failure is hidden and you get this failure transparency. Location transparency talks about hiding the location, the information of location while finding and binding to an object. Internally if you use a naming service and then the internal mechanism say the proxy may find out where the object is and connect to the object. And the actual caller is not aware of the location. Then you may have relocation transparency which marks the changes in location of the object from its client. So location might be changed. If this relocation is also hidden, then you also get relocation transparency.

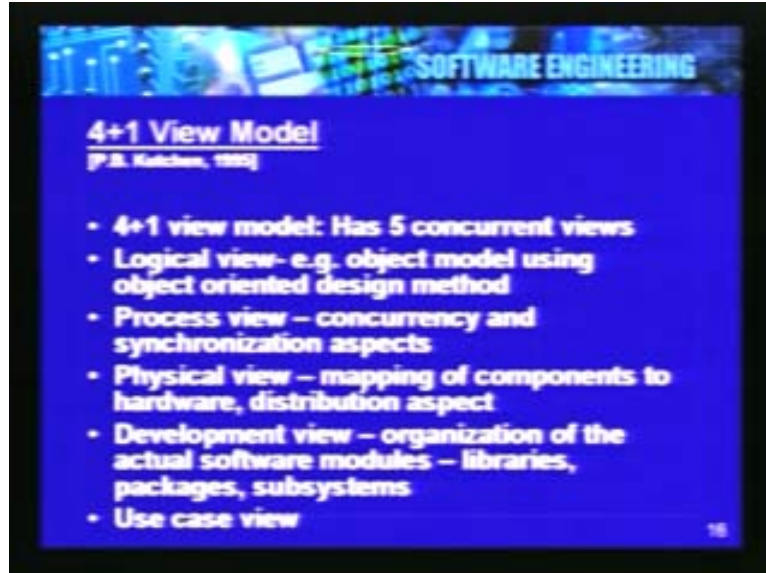
Then there are other transparencies such as migration transparency, replication transparency, persistence transparency and transactional transparency.

(Refer Slide Time: 42:33)



These have to be looked at when you are architecting the system such as what transparencies are you guaranteeing, how are they reflected to your architecture and what are the different components that are responsible to provide these kinds of transparencies. So this ODP approach details all the mechanisms or all the ingredients that go into a distributed systems architecture which you have to use, when you architect a distributed software system. Then there is another approach called 4+1 view model of kruchten. This gives you 5 concurrent views. Logical view talks about the object model using object oriented design methods.

(Refer Slide Time: 43:13)



Process view talks about concurrency, the different concurrent activities and the synchronization aspects. Physical view talks about mapping of components to hardware distribute and then distribution aspects. Development view talks about organizing the actual software modules. This is very important for your deployment. Say you are organizing it in libraries or different packages, say in java you have different packages, in C and C++ you have libraries, how are you organizing them in subsystems? There is also this 4+1 which is 5th view called use case view which is a very high level users view. Use cases have a very important aspect in software development and have been mainly proposed and discussed by Ivar Jacobson and this is also available in the unified process.

So the unified process model approach of architecture is very close to this 4+1 view model. Here the focus is on iterative refinement of architecture. An architecture description is an extract of the model sub system. So if you use UML for modeling your system, you may have many models such as use case model, analysis models, design models, deployment model and implementation model.

(Refer Slide Time: 44:25)



SOFTWARE ENGINEERING

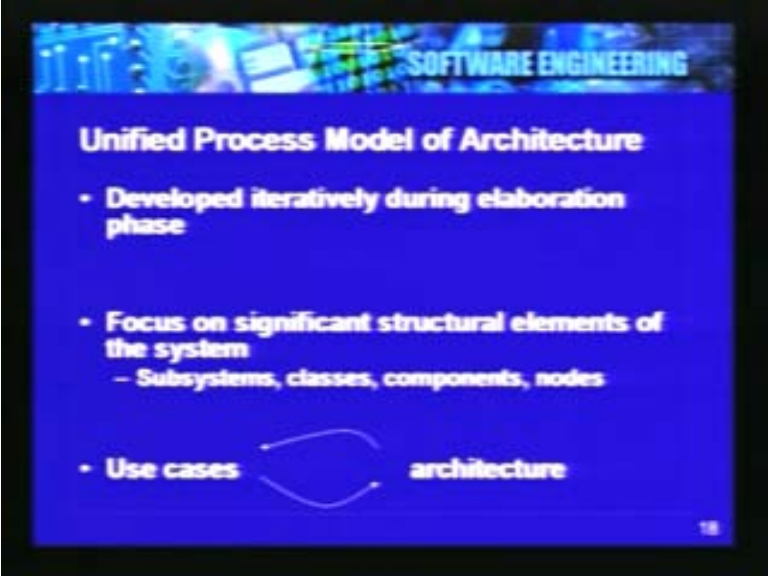
Unified Process Model of Architecture

- Architecture description is a proper extract of the models of the system (use case model, analysis model, design model, deployment model, implementation model)
 - e.g. Contains only architecturally significant use cases, whereas final use case model contains all use cases;
 - Similarly architectural view of design model realizes only the architecturally important aspects.
 - First version of architecture is extract at the end of elaboration phase and so on

17

When you are modeling your system, you extract important aspects from these models and you have architecture. So that is one view point on architecture. This contains only architecturally significant use cases. When you are talking of architectural system you are only extracting architecturally significant central use cases, whereas final use cases model may contain all the use cases. Similarly architecturally important design aspects from design model you can extract. And final version of architecture is extract at the end of elaboration phase of your software lifecycle. So this is the unified process approach to architecture and architecture gets developed iteratively during the elaboration phase.

(Refer Slide Time: 45:45)



SOFTWARE ENGINEERING

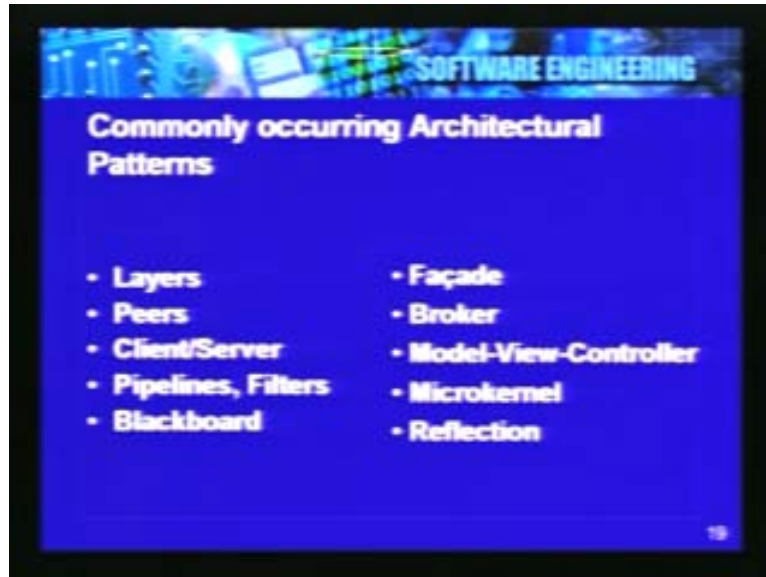
Unified Process Model of Architecture

- Developed iteratively during elaboration phase
- Focus on significant structural elements of the system
 - Subsystems, classes, components, nodes
- Use cases ↔ architecture

18

So the focus for the architectural team is on significant structure elements. You are talking of sub systems important classes and components involved. Here as you refine your architecture, you might refine your use cases and accordingly might refine your architecture. This is another important aspect of architecting systems or this is one approach to architecture. Now we are going to look at some of the commonly occurring architectural patterns which you see in different and almost in all of these architectural approaches.

(Refer Slide Time: 46:20)



We are talking about structuring your components in the architecture. For example you may have layered architecture, you may have peers, client servers, pipelines, filters, black board, façades, brokers, model view controllers, microkernel, reflection and so on. These are some of the important architectural pattern, just as we have design patterns for a design at class level, like how classes are put together through inheritance, aggregation and different relationships.

Similarly this is mainly about high level organization of your modules, sub systems, the structure amongst high level packages or high level entities. These are some of the commonly found patterns in architectural documents or in architectural examples. For example you have seen these layered systems often and you can build a layered architecture of a given system. Here is an example of operating system architecture.

(Refer Slide Time: 47:31)



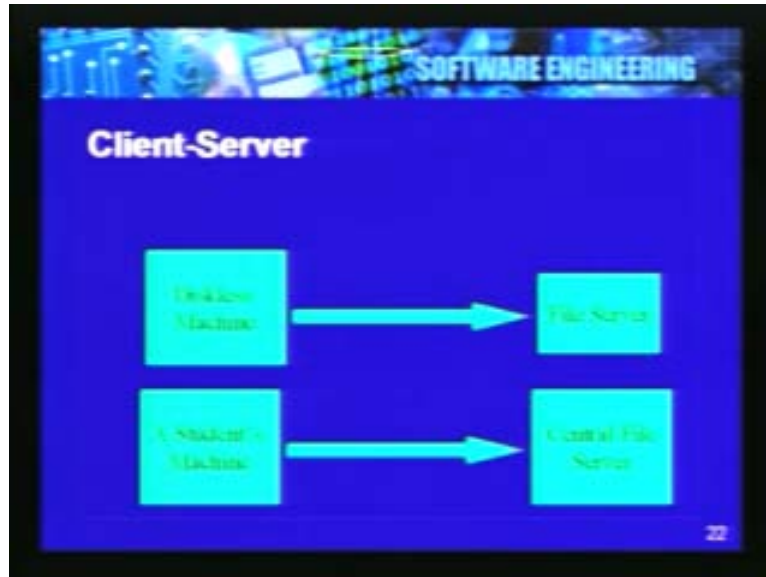
In the top level you have applications which access system calls in the second layer. In the third layer you have the kernel upper half. In the fourth layer you have the kernel lower half and the base layer you have the hardware. This is the layered architecture. The top layer has the higher abstraction than the lower layer. The top layer accesses the lower layer. And any given layer, let us say you talk about system calls, it accesses or it uses kernel upper half and it gives an interface to applications. This is the way to structure layered architecture. You can decompose your system in terms of layers. Another way to decompose your system is in terms of peers. For example you have a LAN on these there are many machines.

(Refer Slide Time: 48:40)



From any machine you can access any other machine. You can deposit files say from machine 1 to machine 4 or machine 4 to 1 or 2 to 3 or 3 to 2 or 3 to 4 and so on. So you have peers which are sharing data amongst themselves. This is peer architecture and not a layered architecture, because you do not have layers sitting on top of each other. You may have client server architecture, where the client component accesses services projected by the server, services offered by the server. Say you have a diskless machine which uses a remote file server or a student's machine which uses a central file server.

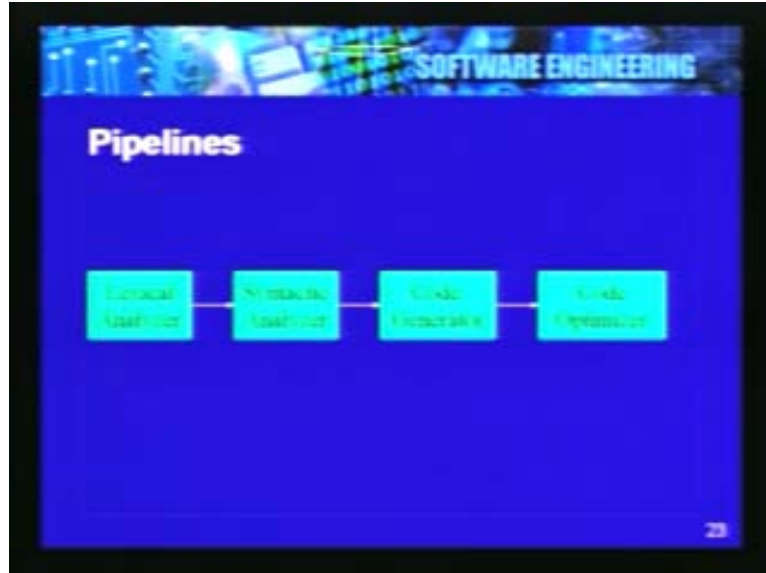
(Refer Slide Time: 49:06)



For example, web services or the web servers. The same idea is applicable to different context. You may have a client server model on a LAN or even on internet which is your web service model or at the low end you may have it even in a sequential program, where you are talking about a client object and a server object within a program. But when you talk about client server mainly, your focus is on distribution aspect and the decoupling through different processes

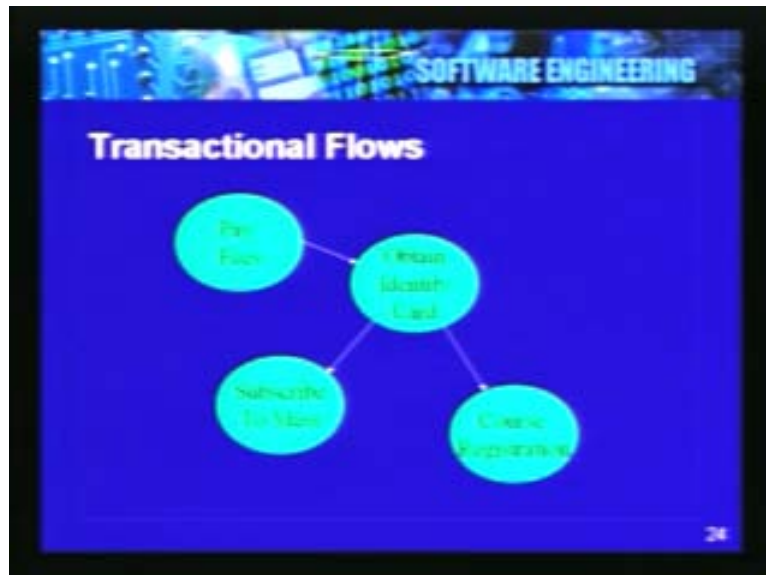
Then you have pipelines where you have stages one after the other. Say in a compiler you have lexical analyzer, then you have syntactic analyzer, you have code generator, code optimizer and so on. These are different phases, so you have pipeline architecture.

(Refer Slide Time: 50:18)



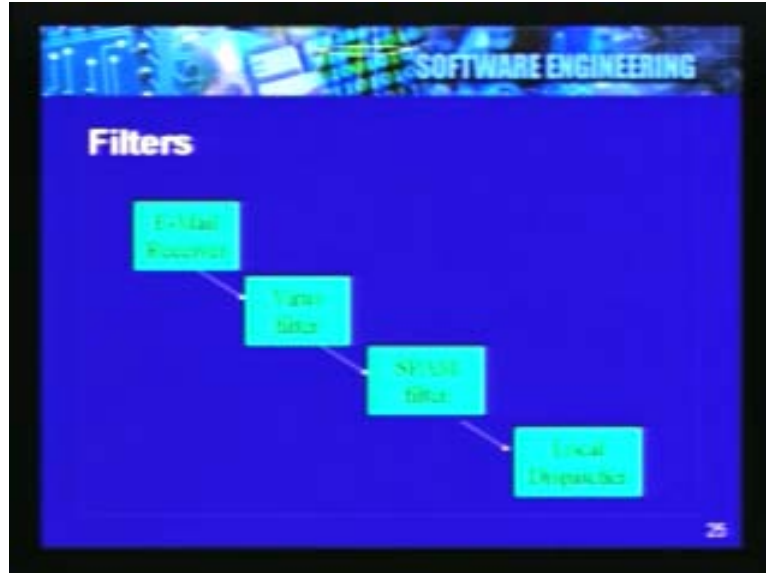
You could have transactional flows. In an academic system, say for example, online academic network in university.

(Refer Slide Time: 50:34)



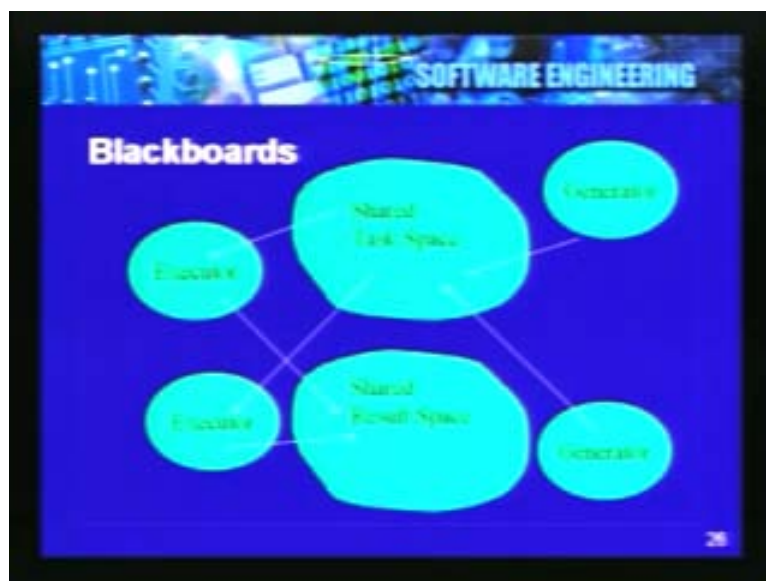
Let us say, first the student pays the fees or you take a note of the fact that the student has paid the fees through the bank and then one can obtain the identity card. After that they have two transactions; one can subscribe to the mess or go on to course registration simultaneously. So you have these transactional flow and dependencies. Then you may have filters in your system.

(Refer Slide Time: 51:14)



Say you have an email receiver, there is a virus filter which filters viruses from attachments and then you have a spam filter which filters unwanted emails or spam and then you have a local dispatcher which dispatches the email received at the local dispatcher end to local machines and users. So you have filters in your pipeline. Some of the components in your pipeline are filters not transformers. Here is an example of blackboard architecture.

(Refer Slide Time: 51:53)



Say in a parallel computing scenario, where you are having a shared memory, you may have a shared task space in which the task generators deposit task and from which the task executors pick up the task. They execute them and then they deposit the results on to the shared results space. We have this shared task space and shared result space as two different black boards which are used by generators and executors simultaneously. This is a shared memory or shared space architecture.

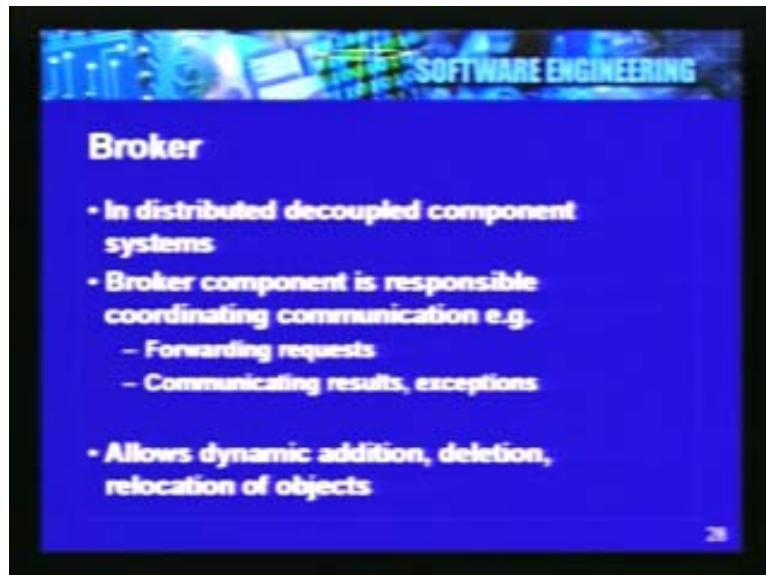
Then there is another architectural pattern called façade. You have a sub system in which there are many classes. This is could be a package and the interface of the package is given through one class called front end and the internal classes are hidden.

(Refer Slide Time: 52:34)



This design pattern is also important from class level architectural point of view. So façade is an important architectural pattern or the way to organize your component architecture. Then broker is another pattern which is seen in distributed decoupled component systems. The broker is responsible for coordinating communication amongst different components. It can forward request from one component to another.

(Refer Slide Time: 53:02)



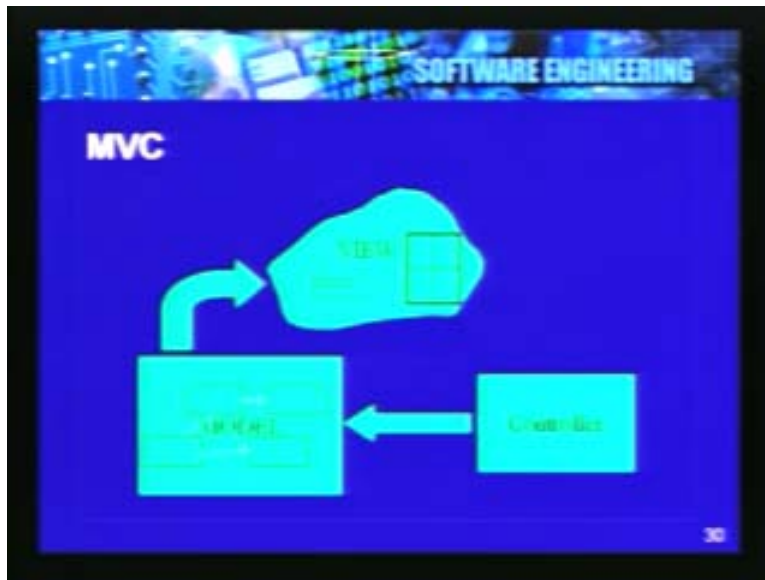
Or it can get back the results, the results are dispatched back to the calling components. Along with the results there could be exceptions or only exceptions and errors which also need to be passed back to the callers. So this broker architecture allows dynamic addition, deletion and relocation of objects, which we will see from this slide. For example you have this broker sitting in the middle and you have the client and the server component.

(Refer Slide Time: 53:50)



The client accesses the broker through proxy and the broker sends the calls to the server through the skeleton. So if you change the server, the client still is not aware of this because it goes through the broker. There is no direct connection to the server. The results from the server go back to the client through the broker. This broker architecture you can see in distributed system such as CORBA. Then you have MVC architecture which is Model View Controller.

(Refer Slide Time: 54:25)

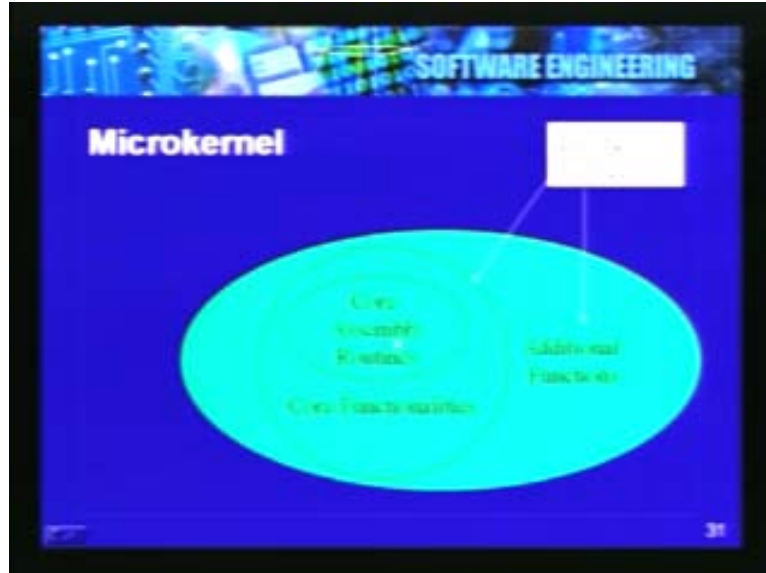


You have a model which you can see on left bottom corner. These are your different entities or your data. And you have a view of this and you have a controller which makes changes to the model. So the changes to the model must be reflected through different views. You can apply different collaborations pattern between the view and this model. For example the view might periodically check if the model has changed or on every change the view may be notified. So there could be push/pull models applied for interactions or for consistency among model and the view.

This is MVC architecture.

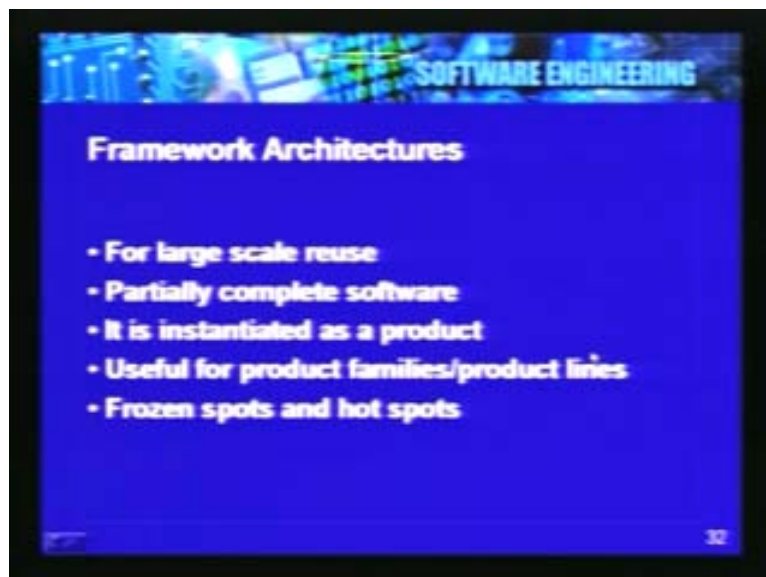
Then you have microkernel, where you develop higher functions on top of view. First develop very low level routine, say for example you have microkernel operating systems where you have **some routines** and on the top of which you can develop core functionality and then you can have additional functions.

(Refer Slide Time: 55:10)



The calling application might use core functions and additional functions or both. But you have a small kernel on top of which you can develop higher level or more detailed facilities or you can extend your functionalities through additional functions. This is microkernel type architecture. Then another important approach to architecture is frameworks. So frameworks are almost partially completed applications. Framework architecture is very important concept in large scale software development or large scale software reuse.

(Refer Slide Time: 55:46)



A framework is instantiated as a product. So for a product family, the framework is very important. Say if you are specializing in library information systems or academic systems or banking systems, you may already have some concrete classes, partially implemented classes for that domain. Now for a given application, for a given customer you may specialize this framework into a specific product. You have frozen spots and hot spots which are projected in this diagram. On the top level, you can see concrete classes in this framework and then hotspots are abstract classes or abstractions.

(Refer Slide Time: 56:35)

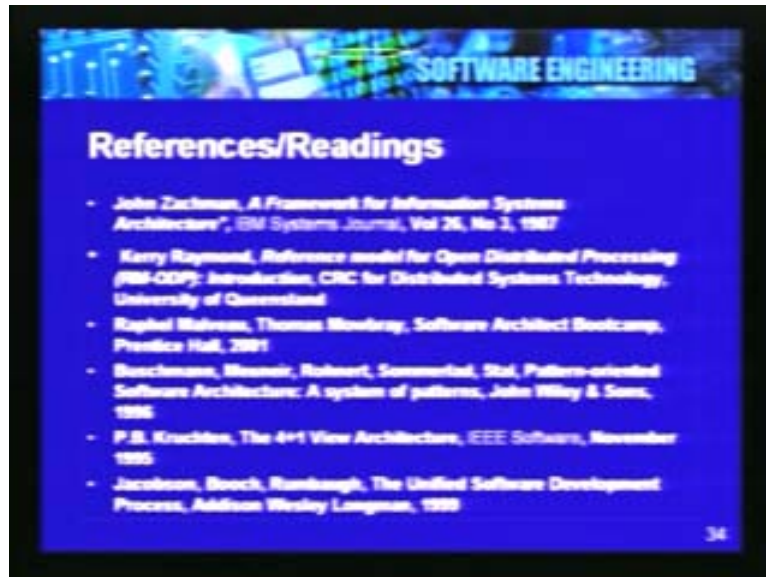


Concrete classes in the framework make calls to hot spots and the hot spots are not implemented in the framework, they are to be specialized. The specializations can be seen at the bottom. So the specializations plug implementation to hot spots and the concrete classes uses the hot spot. The framework implements concrete classes and the hot spots are to be implemented, when you actually specialize this framework to get a product.

So the specializations implement hot spots and concrete classes use these hot spots or abstractions. Say for example if you take graphical editor in which you may have shape as an abstract class and the actual shapes may be specialized and the framework, the editing facility such as copying of shapes or moving of shapes, rotating of these shapes can be implemented already in concrete classes. So a specific framework, say drawing a UML diagram or drawing DFD's or drawing flowcharts can be specialized in this product family through these specializations, where as the application is already partially implemented through concrete classes and hot spots. This is an important architectural concept frameworks which gives you large scale reuse or large scale benefits from reuse in product line architecture.

Now on the last slide we have listed out references and there are many more related research papers and books and you go through some of these and get more detailed perspective on architecture and you could put in practice.

(Refer Slide Time: 58:16)



So architecture is very important from overall point of view of the system, so that the system becomes understandable and then you can base your development on your architectural decomposition.