

Software Engineering
Prof. Umesh Bellur
Department of Computer Science & Engineering
Indian Institute of Technology, Bombay

Lecture - 28

Reuse Continued – Frameworks, Product Families, COTS, Design Patterns – Design Reuse

In the last lecture we were taking a look at the process of software reuse. We looked at what exactly is reuse and what are the different factors that affect reuse; what are the benefits and disadvantages of software reuse and we saw that the main benefits to be **group** from reusing software at various levels. The first one was that of reliability so the components or the cots software that you are reusing essentially would have been well tested in previous situations possibly previous production situations and the bugs could have been I **and doubt** so the problems could have been fixed so the degree of reliability you get by reusing such a component is fairly high.

We also saw that they was reduced process risk because of the fact of reuse so you do not have to get involved in that much more of the estimation process given that this component already exists and simply be reused as is independent project and an obvious byproduct of the whole thing is accelerated development so you can do rapid application development with components stringing them together and just providing the loop between these components and certainly the compliance to standards may come for free because largely components that are meant to be reused are cots software or framework that are meant to be used that typically those that are here to certain standards. So the level of compliance for these standards within your own software goes up by that much.

The disadvantages of reuse clearly are that evolution of your software may not be simple so maintenance becomes much more expensive because of the fact that the components that are in there may not evolve along the direction that the application which is made up of these components is evolving. Certainly there is also the issue of not invented here syndrome where programmers would like to primarily develop their own software. So it not much of a disadvantage as it is a factor that affects reuse and it is that unfortunately very few standards have emerged for the construction of some component libraries worldwide that you can then go and search for when you want a particular component. So, finding and adapting these components may not be a simple task as maybe even developing your own component in some cases. So these are the advantages and the disadvantages of reuse.

When we saw various levels of reuse that could be done and we were into the discussion of component based used the first one was the program generators or the application generators and these helped us reuse the entire algorithm. The second one of the component based reuse where a component was a set of related interfaces, set of related objects that was found together that could be treated as a single unit that provided some piece of functionality and different abstractions or components certainly there could be the functional abstraction, the data abstraction and so on.

Finally we were beginning to look at application frameworks. So there are three other levels of reuse that we have not considered which is what we are going to essentially do in the rest of the lecture today. And the three levels of reuse are the reuse with application frameworks and we will then go into little bit about what application frameworks are, what is the criteria for selecting application frameworks and so on. Then there are product line architectures or cots off-the-shelf systems and reuse based on these systems and finally we will take a brief look at design pattern which helps us achieve design level of reuse.

So what exactly are application frameworks?

We briefly studied this in the last class but application frameworks essentially is the skeleton of an application so it has the infrastructure it has the scaffolding necessary to build an application but it is not an entire application by itself primarily because the fact that two or three things have to be done.

The framework could have to be customized to meet the needs of that particular application development scenario. So, for example, ORB O R B maybe a middleware level application framework that allows you to build distributed object application. What it gives you are certain features such as communication, it gives you the features such as object persistence, transactional library and so on and so forth. But still by itself an ORB is not an application; you have to put the necessary business logic in there for you to create the application. At the same time things like **things like** the Microsoft foundation **classes** is an application framework that helps you build UI and so on. So it can be considered to be the skeleton of an application design that has to be customized and it has a certain number of abstract classes within it or a certain number of abstract components that had to be concretized. So you have to derive, extend and produce the code for these abstract components only then will it become a live application at this point in time.

(Refer Slide Time: 05:43 min)



Application frameworks

- Frameworks are a sub-system design made up of a collection of abstract and concrete classes and the interfaces between them
- The sub-system is implemented by adding components to fill in parts of the design and by instantiating the abstract classes in the framework
- Hence, can be considered as the skeleton of an application design that can be customized by the developer.
- Frameworks are moderately large entities that can be reused

23

There are different classes of application frameworks. The lowest level application frameworks are those that provide communication, operational system features so it may provide user interface, it may provide compilers so you can generate the compiler out of a certain set of classes that exist. So these are called system infrastructure frameworks and an example of this is what is called ACE or the Adaptive Communication Tool kit that came out of Santiago.

(Refer Slide Time: 06:31 min)

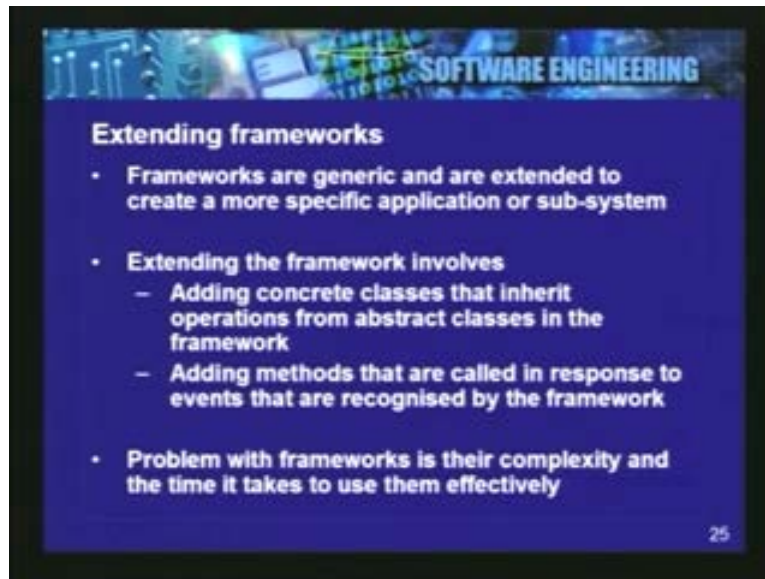


Then this is the middleware integration framework it is primarily the ORBs or they call them as object request brokers or the container architectures that have become very popular today for the construction of enterprise applications and these are basically server side essentially server side containers, standards and infrastructure that help you to build the application on top of this. So, for example, there may be a messaging framework that will allow you to send messages asynchronous to somebody else so the other persons will see messages from a particular **queue** message queue and so on. So the plumbing is all provided in these kinds of frameworks but you still have to write the specific messages that have to get delivered and the processing of these messages is purely your application specific code that has to be written out for this.

So finally there are the enterprise application frameworks that basically support the development of enterprise applications such as telecommunication they maybe vertical applications that are targeted towards a specific vertical domain such as that of telecommunications, such as that of manufacturing, such as that of logistics planning and so on or these can be generic as well so there can be two layers if you will generic application frameworks maybe..... examples of these are things like Apple's NextStep so these are kind of like middleware integration frameworks they give you all the functionality that is required to put together enterprise applications or they may be very specific to a domain. So if you take a look at OMG and **core** bar there are application frameworks around core bar to support the construction of telecommunication

applications only or it can be something like Rosetta.Net for the chip industry semiconductor industry which is basically a framework for B2B integration of the people involved in the semiconductor manufacturing process.

(Refer Slide Time: 08:25 min)



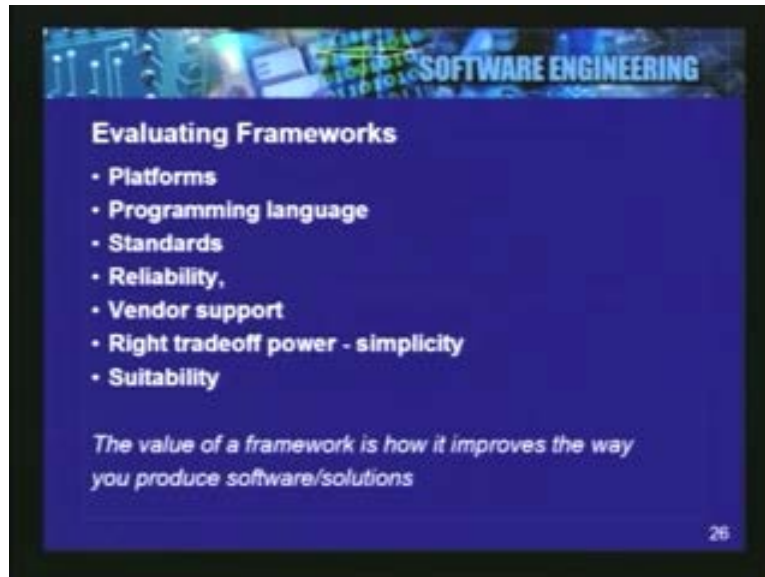
So, frameworks like we said are fairly generic and have to be extended in order for them to be actually useful. The extension has to be done in one of two ways. The first method by which frameworks have to be extended could be to make certain abstract components or classes that exist in the framework become concrete so you have to plug in specific code to instantiate certain object types that are specific to that application. That is one thing.

The second one is typically there are events that the framework will end up receiving and the events would be passed on to the application code for the application code to act upon. An example of this would be a network management or a monitoring framework in which the event that is coming off the network may say that this particular network segment is down; now the framework knows that this event is coming off but it does not know necessarily what you want to do with it so depending on whether you are building a monitoring system or whether you are building some kind of a management system that is actually going to take action on this you might treat this event differently. What you will have to do would be to actually write the code that does the processing of the event in this particular case and so the code would be written that is called by the framework in response to some event occurring so this is the other form of extension that you would likely have to do.

One of the common problems that are obviously faced is that in order to do these kinds of extensions you have to understand the framework pretty well. You have to understand the class hierarchies within the framework, you have to understand the responsibilities of the different modules within the framework and how they interact with each other so the level of complexity may be quite high when you are dealing with frameworks.

Now, there may be several frameworks out there; the question that might come to my mind is how do you evaluate such frameworks in order to make a decision as to which framework to go with or whether you go with the framework at all and here are some of the criteria that you can use to evaluate these frameworks.

(Refer Slide Time: 10:22 min)

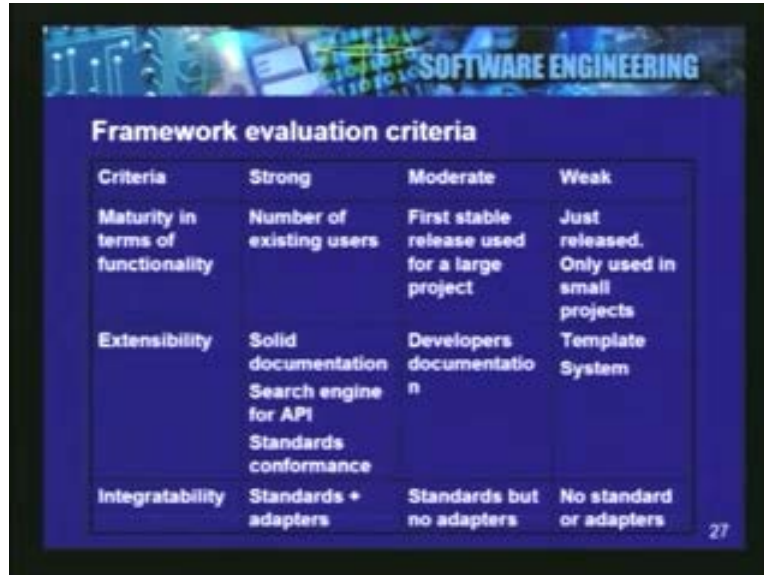


Obviously the different operating platforms that are supported, you know, does it run on Linux, does it run on Macintosh, does it run on Windows machine and so on and so forth. The different programming languages that are used the framework is obviously specific to the programming language unlike design pattern as we shall see later which is merely an abstract notion of design that can be instantiated in any programming language. So in this case you might be looking for things that are only based in Java or based in C plus plus or some mixture of the two; C sharp and so on.

The number of standards that it is adherent to is it only based on standards or is it purely compliant with a fairly high degree of reliability on these standards so that is the question you have to ask yourself and one of the reasons why you ask yourself this question is because you may want to extend this framework, you may want to integrate this framework with other frameworks, other applications and you will not have this integration capability unless it is compliant to the standard; certainly the vendor's support, the simplicity of the framework, the performance of the framework, the reliability of the framework and the suitability for the task at hand.

We can actually take a look at some of these criteria in greater detail. Some of the five to six criteria are what we will look at today.

(Refer Slide Time: 11:40 min)



Criteria	Strong	Moderate	Weak
Maturity in terms of functionality	Number of existing users	First stable release used for a large project	Just released. Only used in small projects
Extensibility	Solid documentation Search engine for API Standards conformance	Developers documentation	Template System
Integrability	Standards + adapters	Standards but no adapters	No standard or adapters

So the maturity of the framework is probably the single most important factor assuming that it is suitable and so on and by maturity what we mean is has this been down several release cycles already or is it the very first release of the framework that is coming out.

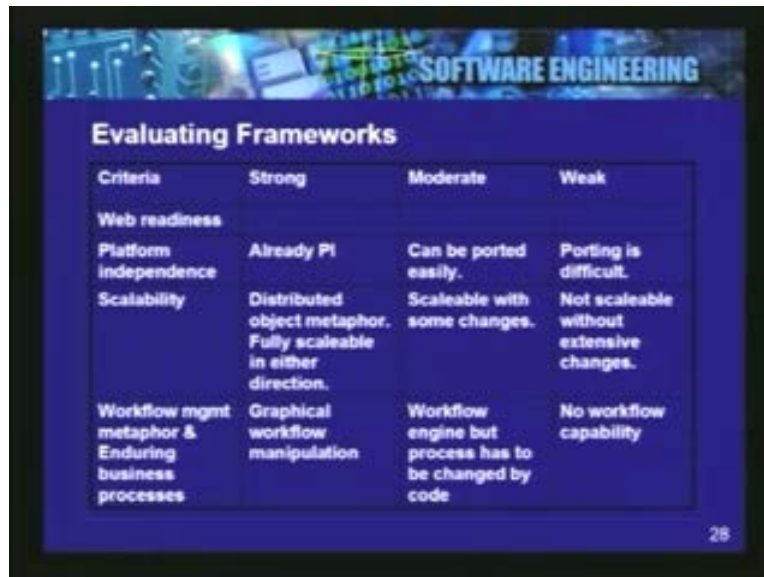
First release products, first release frameworks would have a lot of bugs that you will have to watch out for **you have not been**.....(12:01) because you have not been put through the paces in some kind of production application. So strong framework is something that is pretty mature, it has been through several release cycles and it has been used in several lots production applications. At the other end of the scale is a really weak framework and the area of maturity was typically the first release it may be used by..... some dot releases might have occurred, it may have been used by certain small applications only.

So extensibility of the framework is pretty critical because it would be more often than not it would be the case that you would have to add some capability the framework will not provide 100 percent of the capability. So extensibility would imply that it has solid documentation it has adherence to standards well-known standards that you are familiar with, it may have a search engine for API so instead of sitting and digging through reams and reams of paper you will have some kind of online search capability so figure out the API that you need to work with and so on and at the weak end of the scale there is no documentation at all it is some kind of a template system that only gives you a very small degree of flexibility as far as customization is concerned.

Integrability of the framework is its ability to work with other applications easily or make it work with other applications quite easily. Again this depends on standards; it could also depend on what are called adapters. Adapters are bridges that are built between a particular framework and a specific application. So, for example, there may be a bridge to the IBM NQ series from this particular framework. This would mean that it

will work well with the IBM NQ series of products. There may be a bridge to specific database say Oracle or Sybase so something like that.

(Refer Slide Time: 14:07 min)



Criteria	Strong	Moderate	Weak
Web readiness			
Platform independence	Already PI	Can be ported easily.	Porting is difficult.
Scalability	Distributed object metaphor. Fully scalable in either direction.	Scaleable with some changes.	Not scaleable without extensive changes.
Workflow mgmt metaphor & Enduring business processes	Graphical workflow manipulation	Workflow engine but process has to be changed by code	No workflow capability

Web readiness is something that is being asked of more and more today. obviously the strong end of the scale would imply that this is completely web ready, **it can be** also any extensions that are made can be web enabled pretty easily, the weekend of the scale it has a proprietary UI probably written in C plus plus or something like that and you would need that UI to be installed for this application to run.

Platform independence is obviously a fairly critical feature that you will look for. So it can range all the way from already platform independent so it works on Java or some kind of a virtual machine or the other end of the scale it is completely tied to a specific platform and a significant amount of code would have to be rewritten if this had to be ported on to another platform.

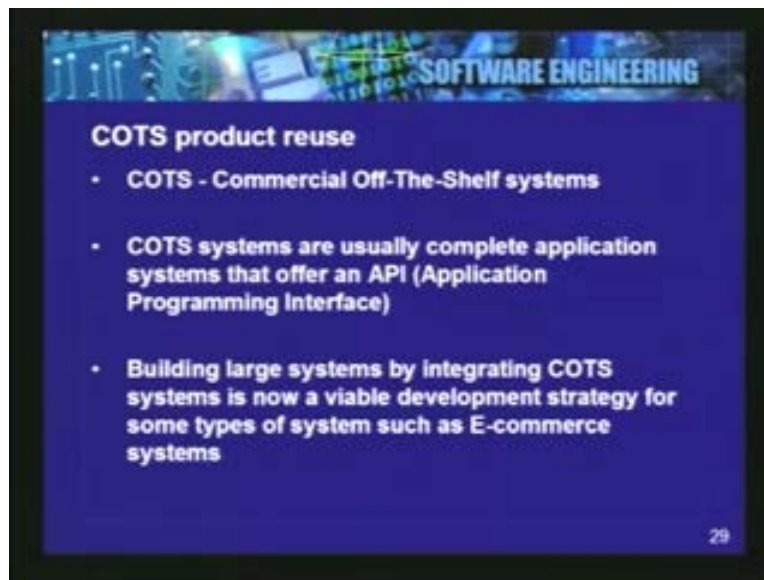
Scalability is another feature and finally the workflow management metaphor; does it have a workflow engine that is in there, does it have any kind of capabilities to graphically define workflow and business processes to this and then subsequently these things can get automated. So these are some of the criteria for evaluating the suitability of frameworks and are frameworks the best form of reuse is hard to say, framework certainly provides a higher granularity of reuse compared to components but that can also be a disadvantage in terms of when you are extending the system the framework does not support it now you might be in a fair bit of problem at this point in time. So it depends on how extensible it is and how you can integrate this with other components and additional frameworks and applications.

Also, the degree of understanding that you would have to achieve in order to make this framework approach succeed would have to be quite high. So you would have to get into

the guts of the framework and understand exactly what these interfaces do and exactly what are the interconnections between these different interfaces because your extension capability depends on the understanding at the end of the day.

So the next level of reuse that we are going to talk about is what is called cots product reuse. 'Cots' stands for common off-the-shelf products. Common off-the-shelf products are essentially entire systems or applications that have been built that you are going to reuse in certain scenarios. Cots systems usually offer some kind of an API but you may and this API would be used for integrating the system along with other systems.

(Refer Slide Time: 16:10 min)



Therefore, for example, if you are building ERP systems for your business you can choose to buy an SAP like framework this is a framework and you have to basically customize the SAP for it to work in this case or you may choose to buy a pre-built application like white plains or something like this which is a complete application in itself so it has the UI, it has the database schema all you have to do is to start populating data into this application it is going to work. So this is obviously a much higher level of reuse but it is relevant in situations where you are building very very large systems; it is not really very relevant in building small things.

So, for example, you are a brand new business and you wish to set up an ERP system for your company, you are a brand new business is pleased to set up an e-commerce web portal for your company. Now this would mean that there is no point developing these some scratch certainly there is some benefit to be gained from using frameworks and then customizing them but these things have become so common, being used so many times now that you might as well by an application that is being pre-built completely and just may be customize the look and feel of this, brand it appropriately for your company and run it as a portal because it may not form a very important part of your company strategy end of the day.

(Refer Slide Time: 17:35)

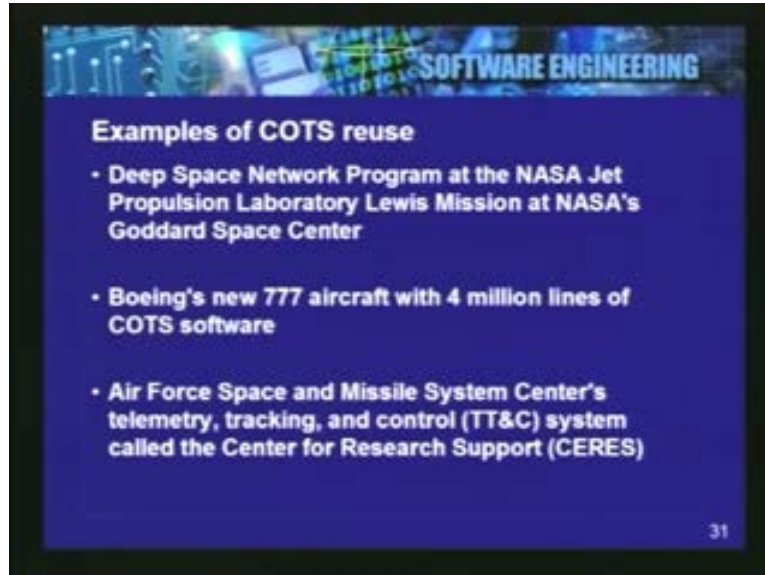


There are different integration approaches that can be taken when you use common off-the-shelf systems or products. The first one is the database approach in which all the operational data is stored in a single central database and all the different applications now will have to access the single central database. This maybe a little difficult because it as assumes that this database schema that you are going to have is something that is accessible by this common off-the-shelf application which may or may not be possible.

The second one is what is called the blackboard approach. In the blackboard approach the data sharing amongst components is basically opportunistic. This means that there is some kind of a middle tier messaging layers, say a public subscribe mechanism which can be used for data sharing. So a system that is going to write data is going to publish data and any of these systems that are interested in this type of data is going to a pick it up. An example could be that instead of integrating an order managements system and a billing system but directly allowing them to talk to each other you can allow them to talk through a public subscribe bus. So the auto management system is going to publish the fact that a new order has arrived; the billing system may pick it up, the inventory system may pick it up and there could be other systems within the company, logistics planning system, shipping system etc which pick up the fact that this new order has arrived and prepared to complete the order; all of them do their tasks and turn.

And finally the object request broker strategy is one in which two applications directly talk to each other. This is not public subscribe, this is not asynchronous messaging but this is synchronous application to application contact that is going to happen. So these are some of the different integration approaches which become very important when you are talking of multiple applications that you are reusing a whole and then integrating or tying then together in order to provide the functionality required by the business.

(Refer Slide Time: 19:40 min)



So, some of the examples of cots reuse; I think the most significant example is the second one on the slide which is Boeing's new 777 aircraft which basically had almost four million lines of prewritten COTS code. So these are entire applications that are picked up and integrated into running of this airplane so this is a very good example largely used in other defense projects and so on.

(Refer Slide Time: 19:58 min)



Some of the problems that you are likely to run into with COTS approaches are lack of control over the functionality is the first problem. Because it is an entire application the granularity at which you are doing the reuse is very very high

so it is a very high granularity of reuse. So there may be, out of the ten features that the system provides you might need four but you may not need the other six that is one level of it so it is a superset of features that you require.

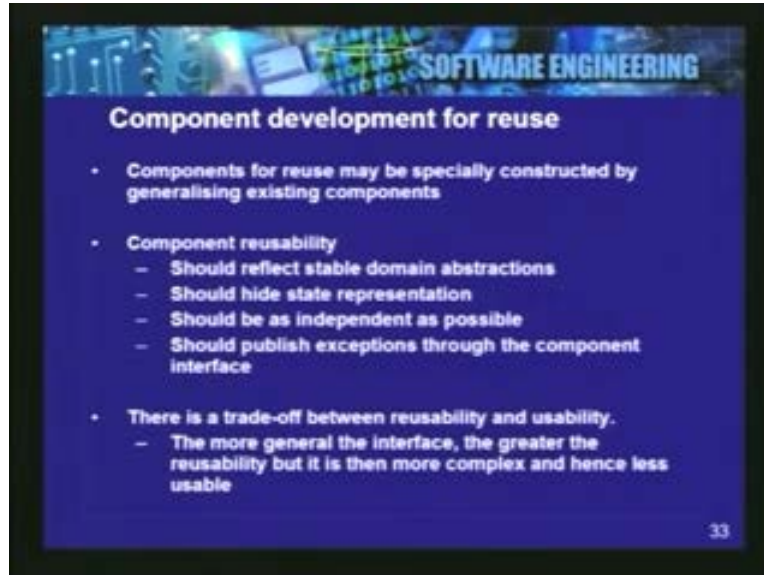
The second situation it may be far worse is that it may have ten of the twelve features that you want but the two other features are not there so you do not have control over the amount or the functionality that you are getting in here because it is a system you have to pick the system as a whole you cannot pick and choose the features of the system; the same thing with performance. Unless it needs the performance needs which it may for some function or may not for some function then again it requires a lot of customization, lot of extension and a lot of recoding in order to get this thing to work right.

Certainly there could be problems in probability; the same problem must be solved with framework but here **intrabobability** becomes much more key because it is likely there are multiple applications working in a single scenario. In the case of frameworks it may be that the framework underlies all the applications and so **entrapolbility** may not be that bigger deal. So they are really conformant with standards, well-known standards, well-established standards it may be really difficult to integrate these things. But the development of technology such as web services and so on **has made us** has allowed us to basically **wrapper** some of these applications with standard interfaces so that they are now able to talk to one another.

The same problem with system evolution as you would see with component based reuse and with frameworks is that every time your system needs to evolve the COTS system should be capable of scaling up and evolving in the same way of sense otherwise this could become a real problem and the support always is going to come from vendors so you are not controlling your own destiny in this case and you have to be careful when you are reusing entire systems that the vendor is going to be around in the next five years for example when your system is going to be around and if they go out of business and there is nobody to support you then how are you going to manage the maintenance and evolution of your system is a situation that you have to keep under control.

So one other thing that we would like to come back to is we saw the notion of reuse with components earlier. One of the topics that we want to close out on that is how do you create components to be reused.

(Refer Slide Time: 22:33 min)



Remember earlier we saw that there were two paradigms your design with use and design for reuse and in this case we would like to take a look at specifically in the case of components because components are essentially things that you create or **unit** encapsulated units of functionality that are meant to be reused in the first place. So we would like to take a look at how do you make a component become more reusable so there is no such things; it is not a black and white, it is not a Boolean thing that this is usable and this is not. Typically the degree of reusability depends on how well **design this component has**.

So let us see the look at what it takes for us to make a component become reusable in the future. **the typically** the general strategy that is used is generalization. So you would take a component that was built for a specific purpose but you would generalize you would raise a level of abstraction and generalize so that this can be used in multiple situations with the API that you are going to provide the component. So the component reusability features are that it should reflect table domain abstraction what does this mean.

So, if there is some feature in the domain that this component is representing let us say if you are in the network management domain it may be representing a MIB which is a Management Information Base for a network device. Now, the notion of the MIB is very standard; the notion of interface with the MIB are well established, there are SNMP standards that existed since decades so there is a certain stability in the domain as far as the MIB is concerned.

But if you take a look for example at the Java management extensions jpi which are now slowly beginning to penetrate the network management domain it is not very stable it is very new, very experimental in nature and is something that keeps changing.

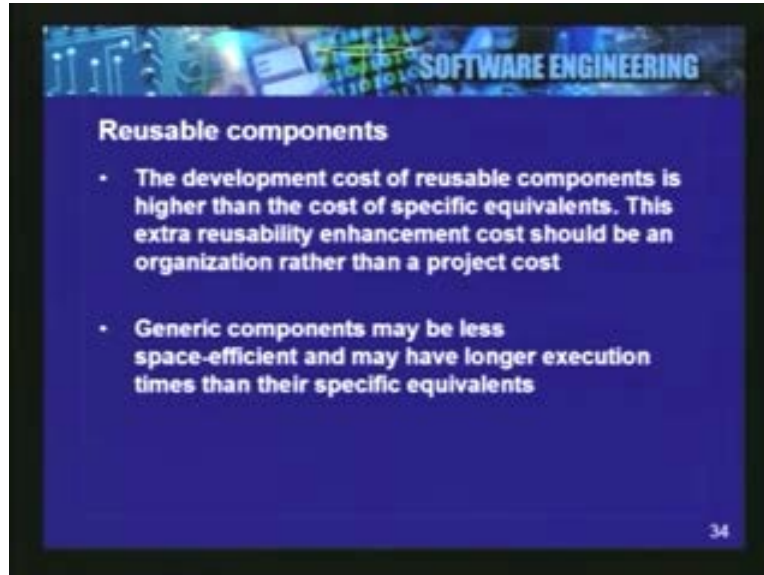
So what we want is really the component should reflect a fairly stable domain abstraction that is when the component will live and will not change by itself every time the domain abstraction changes. It should hide the state representation; the only thing that you want out of a component is what it is capable of doing not how it is going to do it. This is the standard principle of object orientation that we extend into this **into this** scenario. It should be as independent as possible.

So remember, the component has to characterize by the interface that it depends on and the interface that it provides and the number of interfaces that it depends on should be as minimal as possible otherwise this is not a very reusable component because every time you bring this component then you would have to make sure that the other interfaces were available so you might have bring in a whole **host** of other component to support it. And finally it should publish all the exceptions that can occur through the component interface this is very important because there may be situations the component is not able to handle; unless it lets you know that this is exactly what happened and I cannot handle the situation you will not be able to take evasive, corrective or whatever kind of action you need take at that point of time.

So the exceptions have to be well published as part of the components interface also. Suddenly there will exist a certain tradeoff. When you start making components to be more reusable your tending to be it is becoming more generic whereas the component in the first case could have done with say four or five methods four or five functions that performed. Now, to make it to be a more generic equipment you might have to add another five or six functions.

So now the usability of the component because it has ten or twelve functions the understandability and the usability of the component goes down. But it is certainly far more generic; it is applicable in a variety of situations in this case so that is the tradeoff that you would have to make. The tradeoff is that you make it more generic, make it more reusable but the degree of reusability starts coming down that point of time because it is that much harder to understand the component interfaces now. This is something that you have to keep in mind going forward.

(Refer Slide Time: 26:44 min)

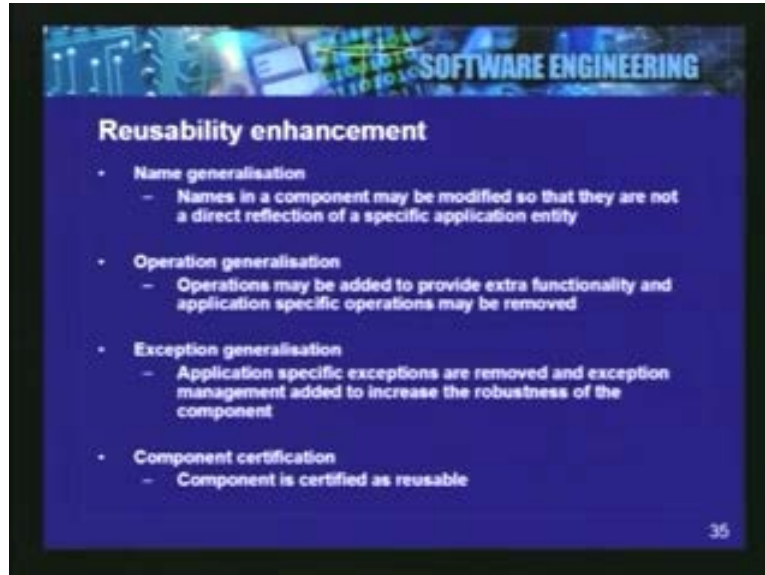


The cost is certainly something that is to be borne in mind because the cost of developing the reusable component is significantly higher than the cost of developing a specific equivalent. What we mean by this is that if you are developing a component or a module just for that one application alone; suppose you are developing a widget for taking a text input in a UI is a good example for component the widget might if it were being written in a very specific manner for that particular application may be that application only require two lines of input for example and you are restricting it to two lines of input at this point in time but this is a specificity this is the specific feature that when you want to make this generic you need to certainly allow for customization of the number of lines of input that can be accepted by this particular text input box widget. So you are making this more complicated but for a reason so that it can be reused across multiple scenarios.

Now this comes out of cost so you have to add the configuration interfaces otherwise you have to test the configuration interfaces and all this is going to cost both in terms of time and in terms of money. So how this cost gets absorbed; does it get absorbed as cost of the project or it gets absorbed as part of the organizational cost that cuts across multiple projects because its component is now reusable across multiple projects is something that you have to keep in mind and if there is a right encouragement within the organization to develop reusable components then this will be a successful project going forward else typically reuse and design for reuse is likely to fail.

So what are the kinds of enhancements that would have to be made to components in order to make them reusable going forward is a question that you would like to ask yourself.

(Refer Slide Time: 28:41 min)



We talked about generalization is the key strategy that is be adopted when you make the component reusable. So what are the different types of generalization that can be done and what are the details of these. So, the first one is generalization of names. Names in a component can be modified to so that they are not in direct reflection of the specific application entity. What this means is that in the case of a text box let us say that the application could have been developed was a telecommunication's application. A telecommunication's application for a particular company, for a particular department had this widget that you are developing text box input widget.

Now, if you name the widget as something that is very specific to that application because it belongs to part of that application's package then it loses some of the generosity so you might just want to name it a text box widget and you may provide an inheritance hierarchy that is completely generic in nature; it should not be inheriting, it should not be..... so anything along the path the naming path that it contains should not be specific to that particular application that is the first thing.

The second one is one of operational generalization. This can be done in two ways. One is specific operational interfaces, can be extended to make it more generic. For example, if you are taking in let us say two parameters to a specific operation and in order to make this operation be generic you might want to take in a third parameter which always defaults to a particular case as far as your application is concerned. Therefore, the strategy to deal with it is that you add the third parameter but you add a default value for the third parameter so that when it is not specified the application or the component always assumes the default so that way you have provided for a generic scenario which can be used by multiple people but that specific application for which it is being developed does not need to have the overhead of specifying the third parameter every time it is going to use the component so that is one dimension along which operational generalization can be done.

The second dimension is one of adding different operations. So the example that we talked about earlier was that of a list or a stack so instead of creating a specific less structure that is only going to be used for that particular application you can create an abstract data type which can be a link list data type or doubly link list data type and this may need many more operations that can be used in different scenarios. A good example could be that let us assume that in your application you are always going to append to the end of the list there was no question of inserting into the middle of the list.

So you really did not need a function called to say insert into a list at position N but in the case that you want this to be a generic component you certainly want a function called that says insert into the list at position N, also, you want another call let us say append to the list that is it is at the end of the list. So here is an example of operational generalization where you want to add certain methods so you may also need to have some methods for initialization specific kinds of initialization that can be done so multiple constructors might have to be provided to be an object interface. Different ways of dealing with instruction in the component would have to be provided; cleaning up of the component would have to be provided and so on and so forth.

The third one is the exception generalization. So here many more exceptions that could occur in different scenarios which would might have to be added. So, for example, component is to be used only in a local access method or it can be remotely accessed from some place. So, if you want to make this a generic component you obviously need to add exceptions that deal with remote communication exceptions that may occur on the network because now the component can be accessed remotely which means it can be accessed in a distributed manner over the network.

So, to increase the robustness of the component, to increase the situations that this component can handle you essentially have to make sure that application specific exceptions are removed and exceptions that are fairly generic and exception management **is taken** is paid in a pretension to increase the robustness of the component.

Finally, one of the reusability enhancement features: this is not really a generalization feature; this has to do with reusability enhancement in general; is that there can be a process of certification. So, for example, if you are providing core bug components you are providing components that adhere to the core bugs standards then you might want to have it certified by O and G which is the organization that deals with the core bug standard. So, component certification could be fairly important because it will also give a certain level of trust **to this** to the user of this components saying.... Oh..! it has been certified by Sun, it has been certified by this authority and I can therefore safely use the component in my application knowing that it has met certain quality standards, it is fairly robust **you know** it is not going to destroy my application going forward and so on and so forth so there may be vendor support that is possible with this and so on and so forth. So, that is something that has to be kept in mind as well.

What is the reusability enhancement process itself look like?

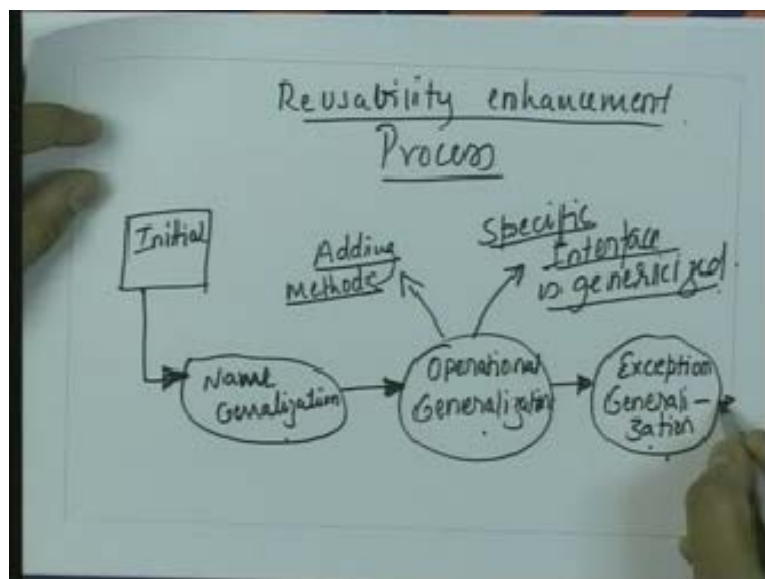
What we have seen so far essentially is that components can be built for reuse and component applications can be built by reusing components. So what we have been discussing here is the process of reusability, is a process of increasing the reusability of the component itself. Earlier we had also seen a similar which dealt with reusability within the software engineering life cycle. How do you account for reusability within your application development life cycle? Now let us take a look at the reusability enhancement process.

The first thing is that you have an initial component interface that you start from. You have built a component and what you need to do is to make this a generic component the component you have built may have been for a specific project so the first thing that you have to apply is the name generalization process. there are plenty of people, for example, whose names include that of the package for which it is being built, whose names reflect the company for which the code is being built, whose names reflect the application and so on so you should try and stay away from that as far as possible so name generalization is the first thing that is done as part of this process.

The second one is operational generalization. And operational generalization as we have seen before can take two different forms: The first form is that of adding methods. By method we are referring to the object oriented term; adding interfaces would be a better way of describing it. One is a specific interface is **genericized**. So these are the two different ways of doing operational generalization.

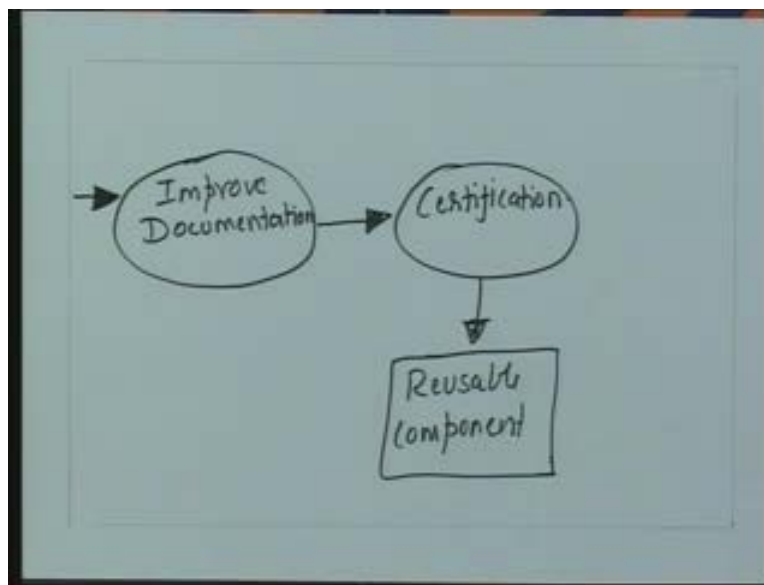
The next thing you may have to think about is the exception generalization. You have already seen these three things: the name generalization, the operational generalization and the exception generalization process.

(Refer Slide Time: 37:02)



Once you pass these steps one of the things that you would have to then do is to significantly improve the documentation of the component because if it is a component that you are going to use purely within your own project and you have written the component then you probably do not need a whole new lot of documentation. But, on the other hand, if somebody else is going to use this component then they would need a lot of documentation because they may not necessarily understand your intent when you develop the component. Finally it has to go through a certification step which will check all of the above: is it generic enough, does it have good documentation, is it robust enough, does it pass all the test that it says it is going to pass and so on and at the end of this process then is going to come a reusable component.

(Refer Slide Time: 38:09 min)



A good example of this might simply be data types. the string data type for example started off fairly with a minimal interface but then as things went along people realized that this was a very very useful data type to have and a lot of functions were now added to the string data type and the final form in which you see the string data type is nothing like what it is used to be when people started working on it and however it is very very useful in a very wide variety of situations. It has quite a few interfaces to it. Obviously could be the source of some degree of confusion but at the same time its usefulness overcomes that confusion because of the fact that it is usable in such a wide variety of situations.

So some other generalization techniques are..... and these are specific techniques that we have come up with. The first one is to identify design decision that maybe abstracted, parameterized or otherwise undone.

(Refer Slide Time: 39:02 min)



Identify dependencies or commitments to remove: So anything that is the dependency that the component has on other components must be removed as far as possible; you want this to be independent. Any hard wide external dependency is what this talks about, make it parameterize so parametric context can be added; this can be user-defined and parameterized dependencies can then be created if you need it to be then large effect operations and objects with well-defined set of small effect operation should be done. This basically says that if there is a single method now single interface that does a lot of things within the component you try and replace this with a set of smaller interfaces which do smaller amount of things.

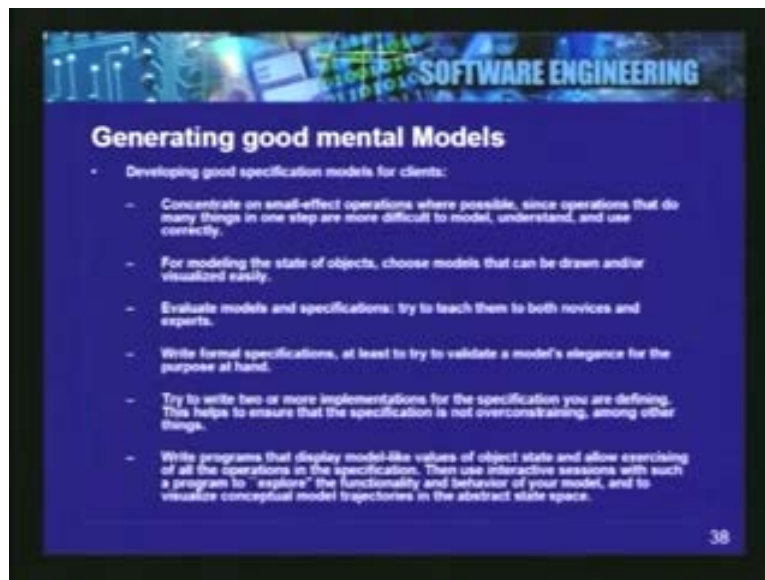
So, for example, there could be one interface on list that says append to the list and increase the count of the elements in the list and so on and so forth but **there could be** it could be replaced by two single operations: one that increases the count of elements in the list and one that just appends the particular item that is being added to the list so that is because the understandability of individual operation goes up significantly and therefore the understandability of the component a the whole goes up as well.

Always write components that have customizable behavior for error condition. this is the notion of exception; writing a lot of exceptions within the interface of the component and making sure that if you cannot handle certain situations then there is a well-defined exception that is going to go back up it is **not to ripple up** and the user of the component can take evasive or corrective action on that particular exception as well.

The last tip here is to generalize some several instances. If the first time you are writing a component, if you say.... Oh..! I can make it generate and immediately go ahead and create a reusable component out of that it is likely that the **situations** for reuse the potential for reuse may not even exist. but if, on the other hand, if you find out that you are using a similar component over and over again in a particular project or across

different projects that is the time to actually create a component out of it, a reusable component out of it. So do not **generalize** it on the first try always wait for empirical evidence that this component is indeed going to be useful across multiple situations and when you run across two or three such situations that is the time to make this a truly reusable component. The other important thing about reusability is to generate a good mental model about the component.

(Refer Slide Time: 41:50 min)

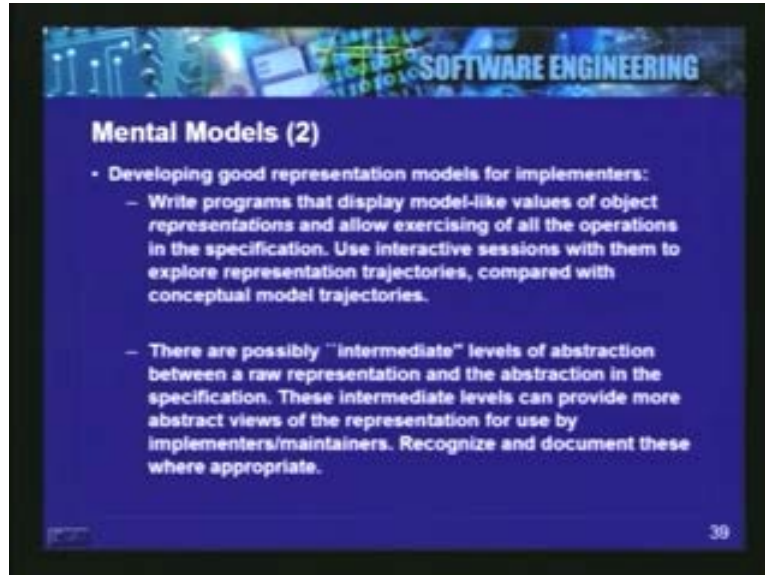


So, the client or the person who is going to use this component in the construction of another application needs to have a good feel for the component that is the only time they are going to reuse it and here are some guidelines for creating good mental models of components as well so specifications have to be very well written for components.

So, if you are just giving a bunch of interfaces with no notion as to what the expected behavior of these interfaces are that is not a complete specification. Remember, in the lecture on **specific** formal specification we took a look at what were called abstract data types and abstract data types contain sections of behavioral specification within it which came in the form of axiomatic specifications. You specified a list of axioms that said that the method interfaces that I am providing in this component or data type have certain rules that they will follow and here are the rules in the form of axiomatic specification. So, if something like that exists it will give the appropriate mental model in terms of how does this component behave when I use it; it is not merely the interfaces.

Also, look at visual representations from modeling the states of object choose models that can be drawn visually so UMS state models, for example, can be used in this case. Writing formal specifications is a very good idea; it both gives the user the confidence that this component has been well thought of and it is going to behave robustly, also, it gives them a very good idea of how this component is going to..... what is the behavior expected behavior if I use this particular interface of the component and so on.

(Refer Slide Time: 43:33 min)



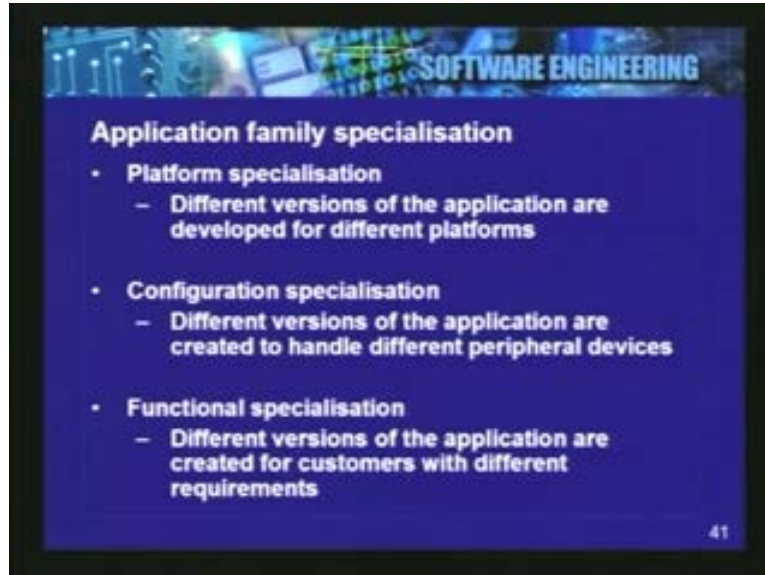
So here are some clues for generating good mental models. Write programs that display model like values of object representations and allow exercising of all operations in the specification. So, interactive sessions are always useful to actually do this and intermediate levels of specifications between a raw specification and an abstraction or something that is also obviously quite useful to do. That is the process of developing reusable components. We are going to move on to the next level of reuse. So, we kind of had a small break in this talk on reuse where we started out by explaining different levels of reuse, we took a look at component based reuse, we took a look at program generators such as algorithmic reuse, component based reuse, then we looked at frameworks and then we diverted a little bit to take a look at how do you create reusable components, not just how do you build applications by reusing components and now we come back to the next level of reuse which is that of application family so product line architectures as they call them.

(Refer Slide Time: 44:39)



An application family or product line is basically a set of applications that drive off a common code. So, for example, it could be driving off of middleware framework for example or it could be something much more specific than that. So, if you take a look at a bunch of ERP applications which may be shipping and receiving and purchasing and acid tracking and logistics and so on all of these are kind of related applications; they all belong to the domain of enterprise resource planning or enterprise resource management so we can work off of a common database, we can work off of common business flows for example which can be reused across these applications and so on so that is what is called a product line or an application family and common core of all of these is reused every time a new application is added to this family and then obviously the application that is added is specialized in some way.

(Refer Slide Time: 45:49 min)

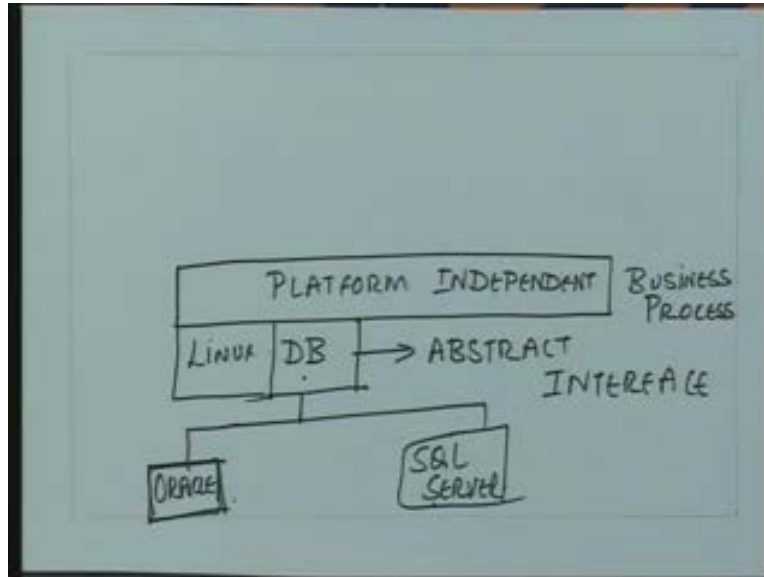


Different kinds of specialization can be done here. Platform specification can be done so different versions of the application are developed for different platforms. The common strategy that are adopted there is that there is a platform specific portion that is abstracted out and then there is a platform independent portion and for the platform specific portions different platform modules may have to be developed.

Here is the example that we will show. So, in the case of an application product line there can be a platform independent layer so this could be for example business process. Business process is completely independent of the platform on which this runs so workflow automation. On the other hand, there can be platform specific modules so there can be a Linux module. It does not only have to be an operating system platform it could be confirming to different databases so there can be a database abstraction and then there can be different databases that is going to sit on top of that I mean under that.

So, for example, it could be ORACLE, it could be SQL server and do on so at this level there could be Linux, Mac, OS and so on. So, the notion is to abstract out the platform independent pieces and the platform specific pieces you create a single abstraction that is used by the platform independent pieces and you create the platform specific versions of those. So here is an abstract interface this would be an abstract interface (Refer Slide Time: 47:31) and this abstract interface is used by the platform independent layer and this abstract interface can then take on two different concrete interfaces or specializations can be done depending on the platform that you going to.

(Refer Slide Time: 47:50 min)



Configuration specialization: Different versions are there; the applications being considered to handle different peripheral devices. So, for example, there may be a web based system that displays inventory but the inventory may be gotten off of a RFID system, it could be gotten off of a **EPC** system and so on and so you may need to have different device drivers for example something to talk to an RFID reader, something that can talk to a hand held device that scans UPC codes and so on and so forth and all of these are different configurations that you would have to end up creating as you go forward.

Finally, functional specialization: The classic example here is that of managing work flow. The workflow could be slightly different in every company. Even though all of them do inventory management, all of them do ERP the actual business workflow could be slightly different and that customization is done through functional specialization.

(Refer Slide Time: 48:52 min)



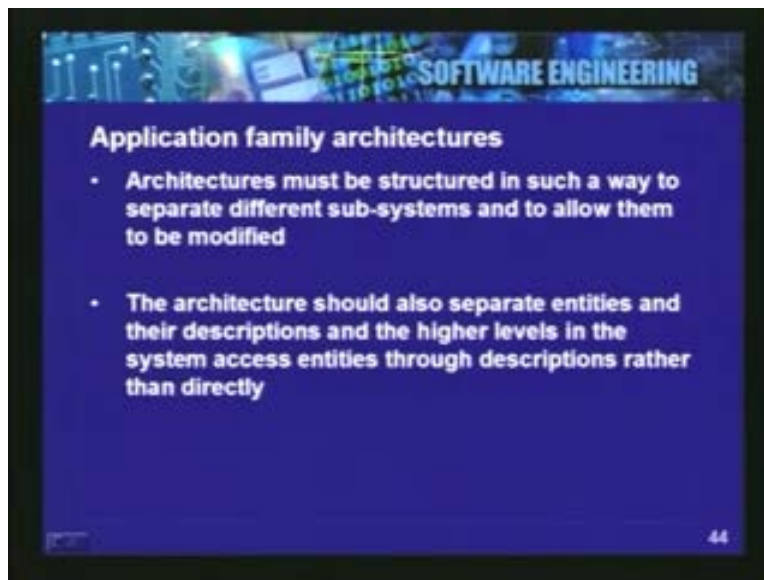
Here is an example of a generic resource management system. See the kind of specializations that can occur. The resource management system has several layers. It has the resource database at the lower most layer; then it has the description of resources and the specification of the kinds of report that it is going to provide and so on and it has different interfaces to add resources, delete resources, query the database, browse the database, administration, reporting etc etc and there can be different layers of access at the top most level namely the user and program programmer access. So, example of a resource management system is an inventory management system.

(Refer Slide Time: 49:44 min)



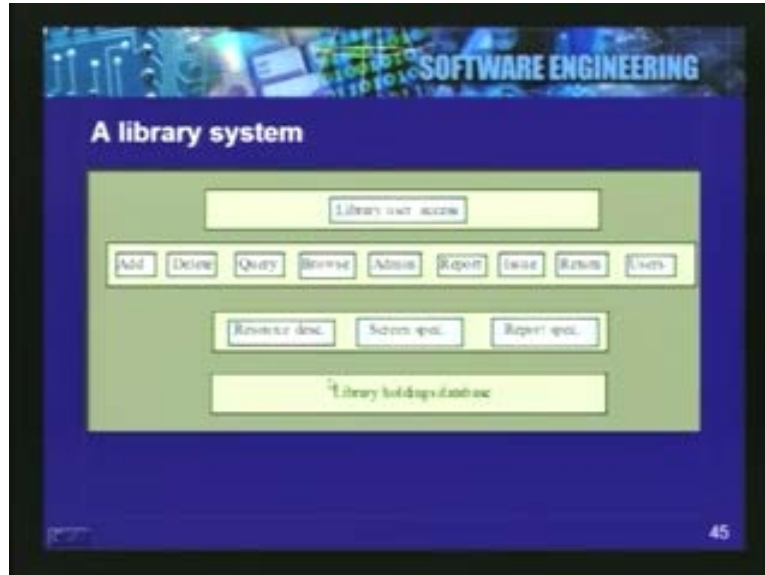
In this case the resource database simply maintains the details of the various inventory that is being managed. The IO descriptions are the structures in the resource database and the input output formats that are used. So, for example, **it can be described** I can extract the listing of the inventory in XML so that it can be passed off to another application. On the other hand, I can extract it into the form report that is created into a PDF file and so on and so forth. The different levels of queries that it provides over this: Are there only fixed queries or are there parameterized queries or there ADHOC queries and so on and different access interfaces. So the key here **is to base the key** is modularization. In product lines or in product application families you have to appropriately modularize everything so that the pieces that can be used or reused across multiple application products are the ones that can be quickly taken and put together.

(Refer Slide Time: 50:40 min)



So the separation of concerns to make sure that the appropriate models or realization is achieved is very very important in this phase. So in this case it is also the library system example in which we have exactly another resource management system.

(Refer Slide Time: 50:55 min)



And in this case the library holdings database is the one that is the set of resources. the same set of descriptions are going to be applied and a set of interfaces or modules are provided; how do you deal with issue, how do you deal with return, how do you generate reports on this and so on and so forth; the set of users so user database might have to be created as well in this case and appropriate user access on what can you do on the library system is the question.

You have seen two: an inventory management system and a library system both of these are resource management systems; both of these allow you to add resources, query resources, delete resources from the database, get reports on resources and so on but both are being used for very different purposes. So the underlying core of both of these systems could be exactly the same as a notion of maintaining the list of users, giving them access levels, the ability to add a resource and so on. But the specific database that holds inventory of let us say manufacturing parts versus holding and inventory of books is different. So the database schema is going to look like slightly different. The details of the resource that is being managed is going to look slightly different.

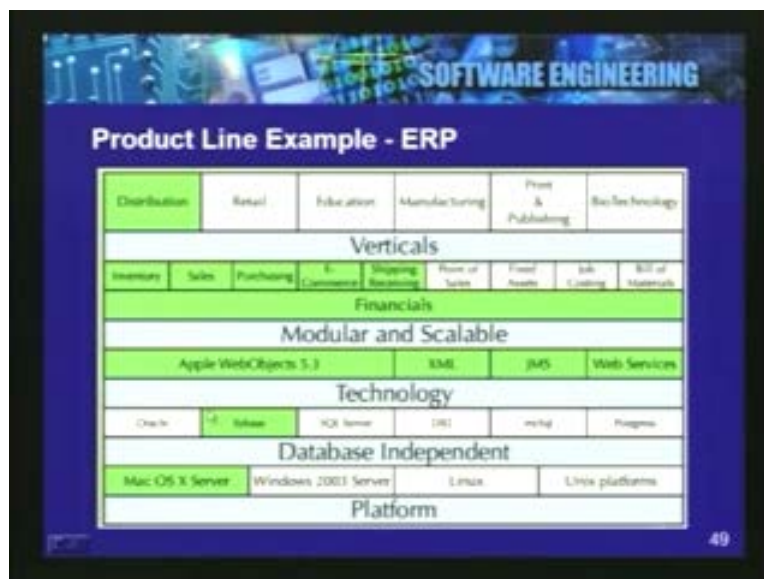
Therefore, in the case of family member development what is done is you elicit the requirement from the stakeholder and then you choose the closest fit family member that exist so there can be several products that are already being built so you pick the one that you feel most closely matches the requirements at that point; you may renegotiate the requirement or you may adopt existing systems or both could be done in parallel and then you deliver a new member.

(Refer Slide Time: 52:33 min)



Hence there is actually incremental change that takes place to member of the family that already exist and that is the key here. So large part of the product is being reused and only an additional small piece is getting done and that is what is explained in these bullet points. Here is another example of ERP.

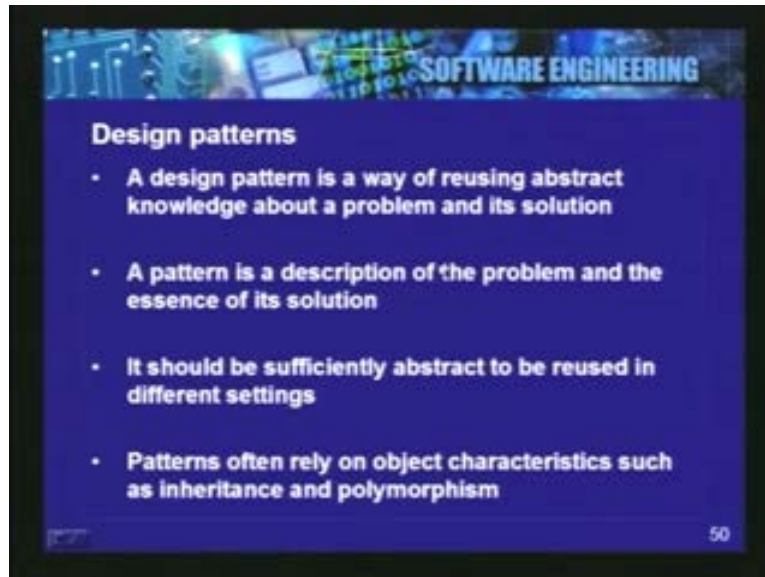
(Refer Slide Time: 52:53 min)



ERP consists of inventory sales purchasing, e-commerce shipping etc etc; ERP consists of inventory sales purchasing, e-commerce shipping etc etc; ERP consists of inventory sales purchasing, e-commerce, shipping etc etc so **you can** all of these share a common core so underlying all these for example this system was built on top of the apple web

object (Refer Slide Time: 53:05) on top of XML, on top of JMS etc so all these in the structure is reused and as you go up; now, distribution for example consists of inventory, sales, purchasing, e-commerce, shipping; retail consists of other things and so on. So that is the notion of products lines.

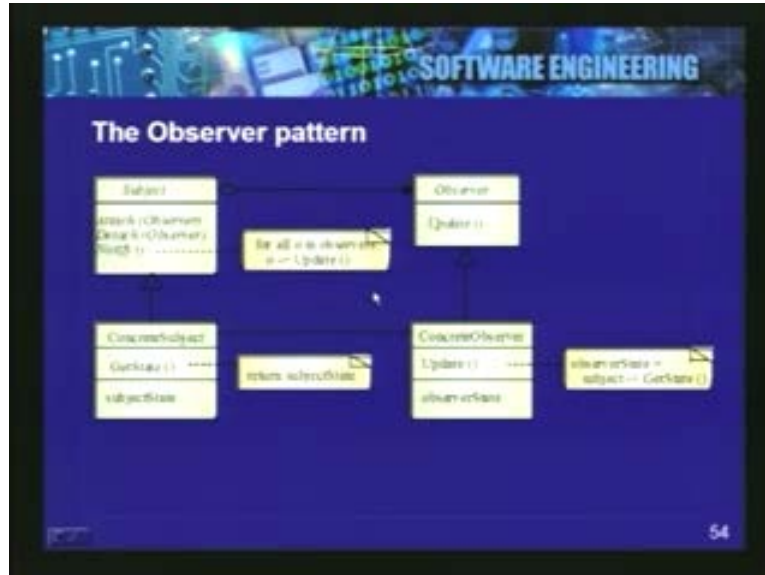
(Refer Slide Time: 53:24 min)



Finally we will briefly describe what a design pattern is. A design pattern is the final level of reuse that we talked about; is the notion of reusing design. So you are not reusing code, you are not reusing product families, you are not reusing architectures etc you are reusing designs and a design pattern is an abstraction of recurring theme that could exist across multiple systems. A good example of this is something like the observer pattern. In the case of the observer pattern the problem that is being defined is one where multiple applications exist that may have to observe a change in the data that is going to take place so an event is going to be passed every time the data is changed and the applications that are going to observe the change in data are observers and they take appropriate action on the events.

Now, this is a very commonly occurring thing. It could occur in the fact that multiple displays may have to be updated, it could occur in the fact that multiple systems may have to take different kinds of actions on the occurrence of an event etc. So why not abstract this out into a design pattern which is described formally using something like UML. For example, here is the observer pattern that is described using UML.

(Refer Slide Time: 54:44 min)



It says that the subject is the one that is changing and these are all abstract classes; these can be concretized at the appropriate place. So this is your example of designed reuse. The notion is being reused, there may not be any code that goes along with it; it just says if you have this particular problem of having multiple observers having to take action different actions on change in one subject then here is the design you may want to observe. This is an example of design reuse where the code itself may not get reused but the notion or the design is getting reused because it is applicable on a wide variety of situations.

Thus, what we have essentially learnt here is two facets of reuse: reuse itself is pretty critical it can save a lot of money but on the other hand you have to be very careful about how you end up employing reuse, also designing your applications for them to be reused, designing your components for them to be reused is another aspect of reuse that you have to be aware of and certain conscious choices and process models have to be followed in both the cases: Designing for reuse and designing with reuse; how do you make a component reusable versus how do you reuse components within building an application. Some of the techniques and strategies that we have gone through in the course of the last two lectures and this is something that is obviously a very very important facet of software engineering in today's application development.