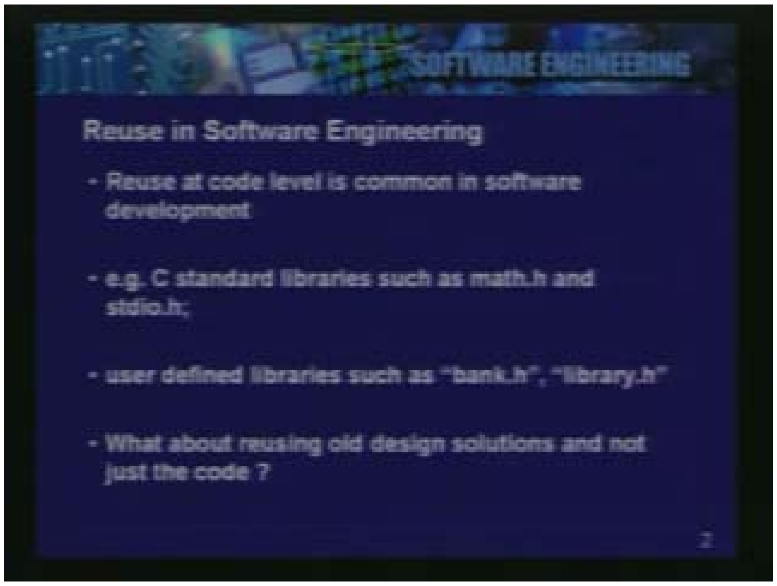
Software Engineering
Prof. Rushikesh K. Joshi
Computer Science & Engineering
Indian Institute of Technology, Bombay
Lecture - 15
Design Patterns

Today we are going to talk about an important aspect of design, which is reusability of design. We are going to see how much our old design can be reused in our new context. A few years ago researchers in object oriented software engineering realized that there are many commonly occurring patterns which repeats again and again in different domains. Even if the requirements are different, the class level design has lot of similarity. They thought why not look at them carefully and document them as reusable pattern. So we are going to talk about these patterns called as design patterns.

They are called patterns, because they repeat again and again. A pattern is something that repeats. This lecture is about design pattern. If we look at reuse in software engineering in general, we can see that reuse at code level is very common. In this slide we can see that reuse in software engineering can be obtained at various levels.

(Refer Slide Time: 02:19)



SOFTWARE ENGINEERING

Reuse in Software Engineering

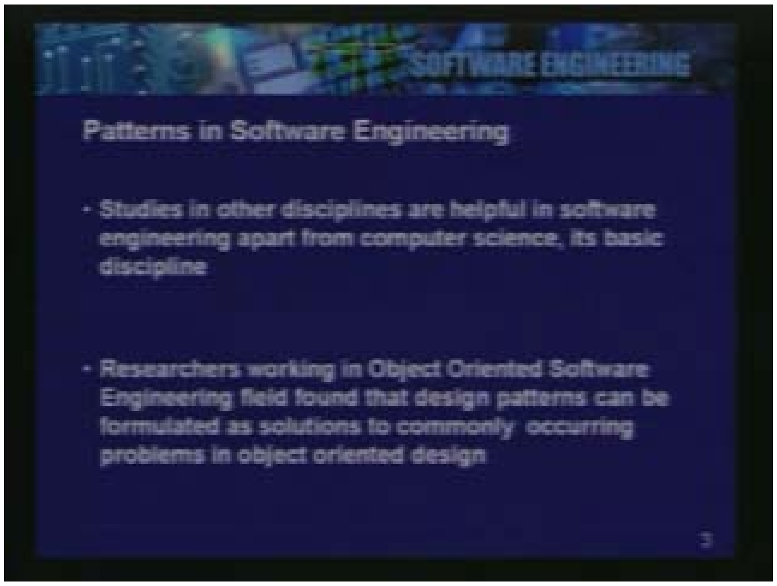
- Reuse at code level is common in software development
- e.g. C standard libraries such as math.h and stdio.h;
- user defined libraries such as "bank.h", "library.h"
- What about reusing old design solutions and not just the code ?

2

The most commonly occurring reuse is at code level. We reuse code, for example the C standard libraries such as math.h stdio.h and GUIs in java or even the user defined library such as bank.h library.h. If you are developing banking software or library software, you might design some of those classes and implement them as readily available implementation and use them in your code. These 'dot h' libraries in C provide the prototype definitions, and the actual code or the implementations of these prototypes are available in their corresponding implementation. The 'dot h' files that we include in the code represent the reusable code.

These are typically shipped by the designers of the software. Along with the language, you get these files bundled with the language environment. This is an example of reuse of code. What about reusing old design solutions and not just the code? We should be able to reuse the design solutions and not just the code. For example we have designed a class diagram a typical way of interaction among the classes. Can we use those results? That is what we will now talk about patterns in software engineering or specifically patterns in design.

(Refer Slide Time: 04:16)



SOFTWARE ENGINEERING

Patterns in Software Engineering

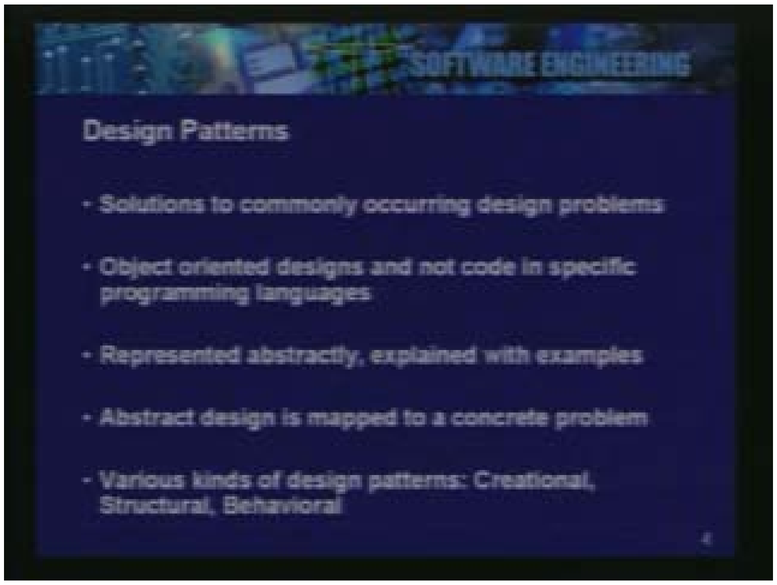
- Studies in other disciplines are helpful in software engineering apart from computer science, its basic discipline
- Researchers working in Object Oriented Software Engineering field found that design patterns can be formulated as solutions to commonly occurring problems in object oriented design

3

Studies in other disciplines are very helpful in software engineering apart from computer science which is its basic discipline. The research in design pattern was motivated after patterns in other disciplines such as architectural patterns. If you look at building architecture, you see patterns everywhere. For example, houses with two bedrooms, three bedrooms, one bedroom, kitchen, basic sitting spaces, the hall etc. All these are patters for building architecture. If you look at temple architecture, there are patterns and there are different kinds of architecture. So there are patterns belonging to every kind of architecture. As you see the pattern in real life and in different disciplines, software engineering researchers also thought that one can have these patterns in software design as well.

Object oriented software engineering has benefited from object orientation which enables us to represent these patterns as collections of classes and interactions amongst them. So what are design patterns? These are solutions to commonly occurring design problem, object oriented design and not the code in any specific programming language. We are not talking about just the reusable code as in libraries which you use through ‘dot h’ files or in java you import adjusting classes reuse them as it is.

(Refer Slide Time: 06:05)


SOFTWARE ENGINEERING

Design Patterns

- Solutions to commonly occurring design problems
- Object oriented designs and not code in specific programming languages
- Represented abstractly, explained with examples
- Abstract design is mapped to a concrete problem
- Various kinds of design patterns: Creational, Structural, Behavioral

4

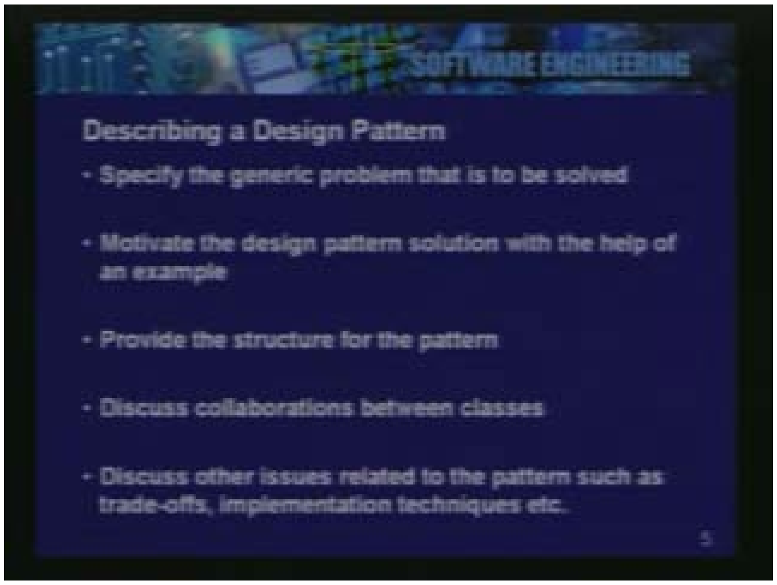
We are talking about design. Typically if we look at design patterns, they involve a few classes may be three four sometimes even two classes. We are talking about collaborating classes. If you have a design of such collaborating classes, we should be able to use that design in a different domain. How do we represent these patterns? They are represented abstractly. You identify roles with every class and then explain them with concrete example and then this abstract design is mapped to a concrete problem. You can imagine a pair of design patterns, say for example, you have a pattern library in which all these abstract designs are documented and when you document them, certain template has to be followed.

When you look at these abstract designs, each class is given a role and it has a specific set of member functions which are defined very generically, so that you can map them to concrete problems which are different. In the concrete problem, you have different class names, different function names etc. These are all very specific to the problem whereas in a design pattern description, you do not have to document the exact name and the exact specifications of a given problem. But it is necessary to document them very generically and then it can be assisted with the help of an example. There are various kinds of design patterns. For example, there are creational patterns, structural patterns, and behavioral patterns.

How do you describe a design pattern?

- First you specify the generic problem that is to be solved.
- Motivate the design pattern solution with the help of an example.
- Provide the structure for the pattern.
- Discuss collaboration between classes.
- Discuss other issues related to the pattern such as trade-offs, implementation techniques etc.

(Refer Slide Time: 08:22)



SOFTWARE ENGINEERING

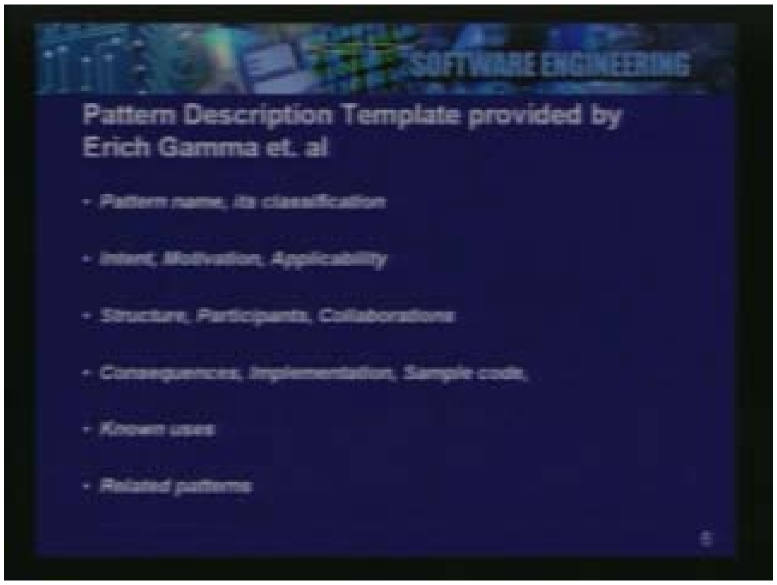
Describing a Design Pattern

- Specify the generic problem that is to be solved
- Motivate the design pattern solution with the help of an example
- Provide the structure for the pattern
- Discuss collaborations between classes
- Discuss other issues related to the pattern such as trade-offs, implementation techniques etc.

5

These are the things describes a design pattern. What is generic problem that you want to solve? If a designer has a specific requirement, he can just go through this list of existing pattern and see whether his requirement matches with any of these generic problem specified by the pattern. Then he looks at the example and sees whether the example is also similar to what he has in mind. If that is the case then he can go through the structure and then implement it in his domain. The specific template followed by Erich Gamma and the team who wrote famous book on design pattern is as follows.

(Refer Slide Time: 09:28)



SOFTWARE ENGINEERING

Pattern Description Template provided by Erich Gamma et. al

- *Pattern name, its classification*
- *Intent, Motivation, Applicability*
- *Structure, Participants, Collaborations*
- *Consequences, Implementation, Sample code,*
- *Known uses*
- *Related patterns*

6

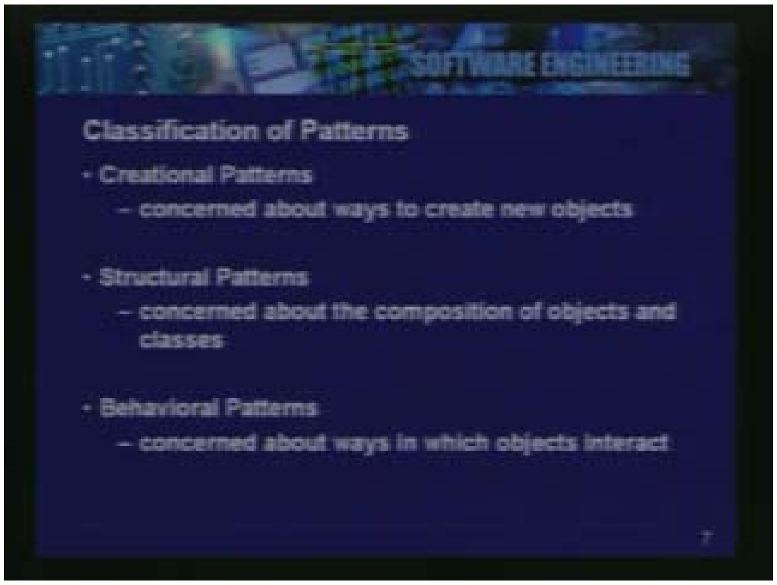
There are many patterns which are described by Erich Gamma and other researchers. There are patterns descriptions given by many researchers. There are few books available and I will be giving their references at the end of this lecture. Coming back to the pattern description template provided by Erich Gamma and team; first start with the pattern name and then classify the pattern. Classification means, we need to specify what kind of pattern it is and what the main focus of the pattern is. Is it a creational pattern or is it structural pattern or is it a behavioral pattern? etc. Patterns can also be classified according to the technological domains.
For example is the pattern meant for distributed systems? Is the pattern meant for real time systems? Etc.

There are patterns in other lifecycle phases as well. For example there are coding patterns, analysis patterns and so on. Then one should specify the intent of the pattern. Which means, what kind of problem, a generic problem that the pattern could solve? Then give the motivation for the pattern and talk about its applicability. Then give it structure, participants, collaboration. Participants are all the classes in that particular pattern and then how do they collaborate with each other? What is the protocol? And then various other important aspects about the pattern are also discussed. For example what are the consequences of using this pattern? Are there any limitations? What kind of implementation it needs?

If you look at implementation of patterns, there could be different independent programming languages and same pattern can be implemented differently in say C, C++ and Java. Object oriented language C++ and Java or Smalltalk. If you implement the same pattern the implementations will look slightly different from each other, because every language has different construct or they vary from each other. For example multiple inheritance of classes, concrete classes are not supported in Java, whereas they are supported in C++. So the implementations vary a bit.

Then you have to also talk about the known use of the pattern. Something becomes a pattern only if it has been used in different domains. There must be multiple uses of it otherwise it cannot be a pattern. You might have a very specific solution in your application domain or in your specific application and you cannot call it as a pattern unless you establish its reusability. You should show its existing known uses. Then you have to also talk about the related pattern. One should be able to distinguish among the closely related patterns and apply or use this specific pattern and should be able to choose it correctly. We have just seen that there are creational patterns, there are structural patterns and there are behavioral patterns.

(Refer Slide Time: 13:07)


SOFTWARE ENGINEERING

Classification of Patterns

- Creational Patterns
  - concerned about ways to create new objects
- Structural Patterns
  - concerned about the composition of objects and classes
- Behavioral Patterns
  - concerned about ways in which objects interact

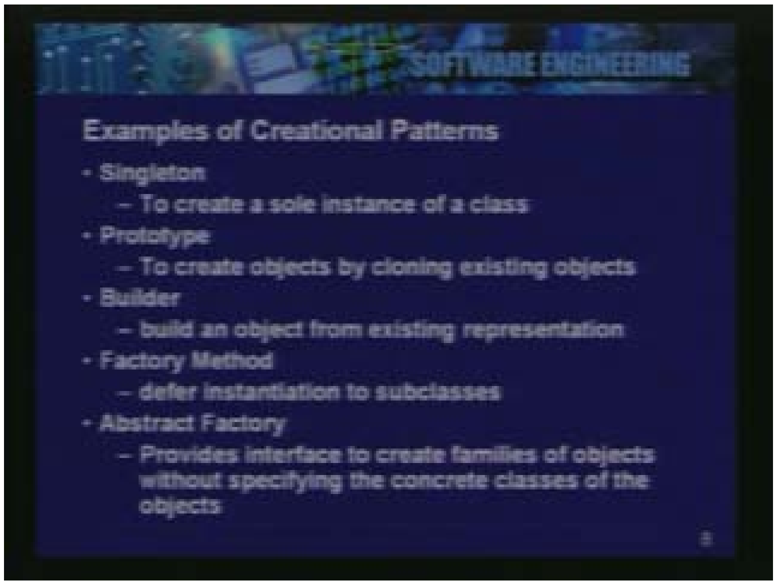
7

The creational patterns are concerned about the ways to create new object. The Structural patterns are concerned about the composition of objects and classes. How objects are connected together and what kind of composition? Basically the main focus is on the structure. The behavioral patterns are concerned about the ways in which object interacts. How do they interact? Is the message sent from one object to another and from that object to another object and so on in a sequence? Or is it a broadcast kind of message? Or is there a handshake protocol? There has to be a structure if the objects have to collaborate. But in behavioral patterns, the behavior or the collaborations are more important.

Similarly in structural patterns there may be collaborations, but the structure is more important and in creational pattern what is more important is the way you create the object. The simplest way to create is simply use the class and instantiate the class. But then there are other ways to create object and we will see some example subsequently. These are examples for creational patterns. There is a pattern called singleton pattern. It talks about creating a unique instance of a class, the sole instance of a class.

That means firstly the class should be able to create an instance and secondly it should be able to create only the sole instance. If it has created the instance earlier, any other request will give you only the old instance. One should not be able to create more than one instance of the class. Then you have another pattern called prototype. Prototype pattern gives us a solution for object creation from a given object. How do you create a new object from a given object? I have an object handled with me but I don’t know what is the class of that object and then how do I create an object similar to the existing object that I have? Prototype pattern gives us the solution to that problem.

(Refer Slide Time: 14:33)



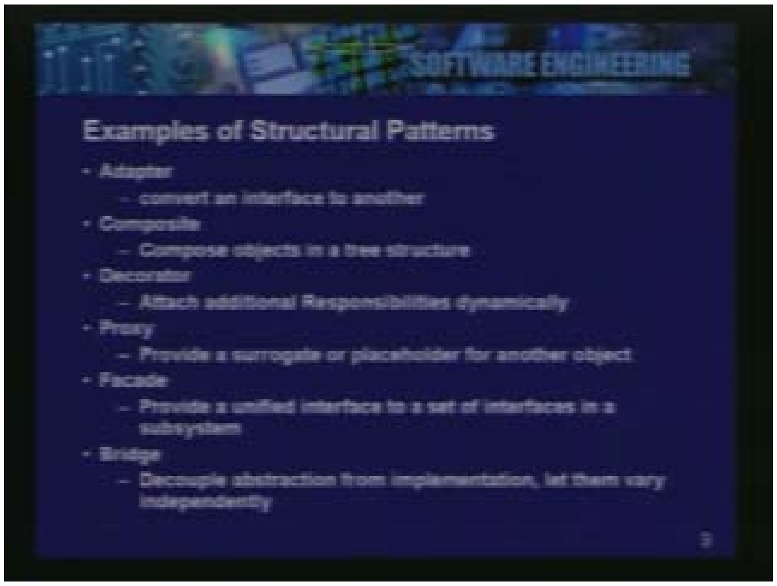
SOFTWARE ENGINEERING

Examples of Creational Patterns

- Singleton
  - To create a sole instance of a class
- Prototype
  - To create objects by cloning existing objects
- Builder
  - build an object from existing representation
- Factory Method
  - defer instantiation to subclasses
- Abstract Factory
  - Provides interface to create families of objects without specifying the concrete classes of the objects

Builder pattern creates an object from an existing representation that is something existing from which you have to create an object. Factory method defers instantiation to subclasses so that you can have polymorphism on creational methods. And abstract factory provides interface to create families of object without specifying the concrete classes of the objects. These are various kinds of creational patterns. We are only going to go through one or two examples in this lecture. Then there are various kinds of structural patterns. Some examples are mentioned on the slide. The first one is adapter pattern. It converts an interface to another. Just like AC to DC adapters, the adaptors in your software tool converts one interface to another.

(Refer Slide Time: 16:26)



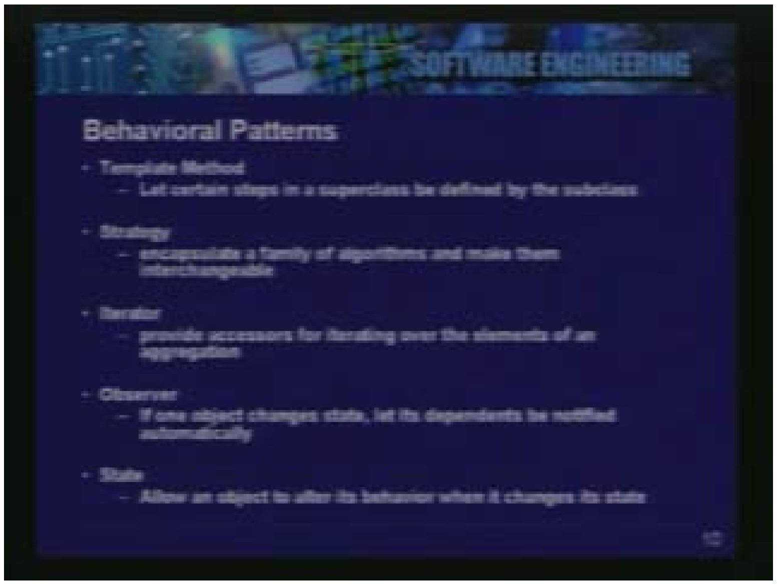
SOFTWARE ENGINEERING

Examples of Structural Patterns

- Adapter
  - convert an interface to another
- Composite
  - Compose objects in a tree structure
- Decorator
  - Attach additional Responsibilities dynamically
- Proxy
  - Provide a surrogate or placeholder for another object
- Facade
  - Provide a unified interface to a set of interfaces in a subsystem
- Bridge
  - Decouple abstraction from implementation, let them vary independently

The composite pattern: It composes objects in a tree structure. Say you have tree structure objects and you want to represent design for such trees, so that you should be able to create an arbitrary tree satisfying given properties. Decorative pattern allows you to attach additional responsibilities dynamically. Facade pattern provides a unified interface to a set of interfaces in a sub system. Bridge pattern decouples abstract from implementation and it lets them vary independently. Abstractions can vary independently and implementations can vary independently. These are some examples of structural pattern. We will again see one or two patterns in this lecture. Now, here are some examples of behavioral pattern. Template method which lets certain steps in a super class be defined by the sub class. Look at the keyword 'steps' in the above statement.

(Refer Slide Time: 17:45)


SOFTWARE ENGINEERING

Behavioral Patterns

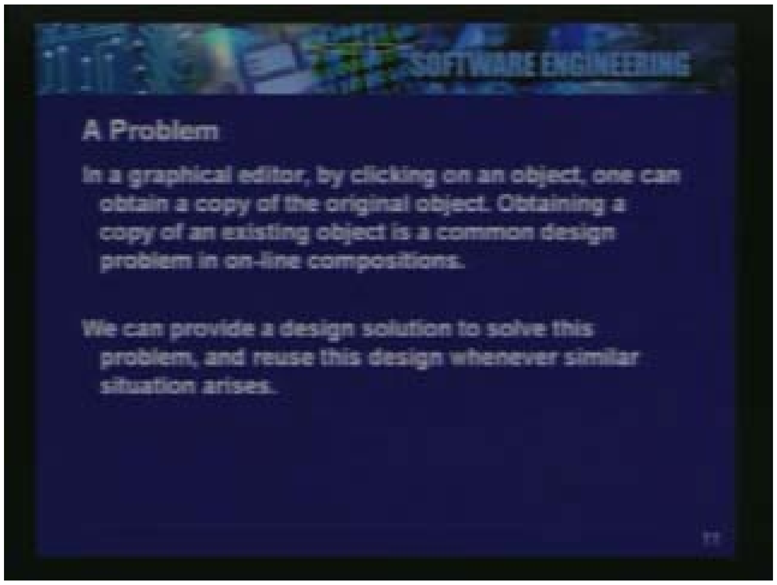
- Template Method
  - Let certain steps in a superclass be defined by the subclass
- Strategy
  - encapsulate a family of algorithms and make them interchangeable
- Iterator
  - provide accessors for iterating over the elements of an aggregation
- Observer
  - If one object changes state, let its dependents be notified automatically
- State
  - Allow an object to alter its behavior when it changes its state

You may have a member function implementation in the super class and some call in that member function implementation or some function calls in that member function implementation in the super class may not be concretely implemented in super class itself. They have to be provided by the subclasses. Such a method that is implemented in a super class in terms of non implemented method which needs to be implemented by the sub process is called template method. We are going to see an example of template method later. Strategy pattern encapsulates the family of algorithm and make them interchangeable. You can apply different strategy at different times.

Iterative pattern provides accesses for iterating over the element of an aggregate. Say you have a collection and you want to take a specific action for all elements in that collection. You have to go over the collection. In order to do that, what is the solution, how are you going to iterate over a given collection? Iterative pattern gives us the solution for that. Observer pattern allows us to design the observer or the producer/consumer kind of network or the event generators and action listener's kind of collaboration. If one object changes the state its dependents will be notified automatically. The observer keeps track of the observed.

And there is another pattern called state. It allows an object to alter its behavior when it changes its state. These are some examples of behavioral pattern and we are going to look at some of them in this lecture. Now let us start with a concrete pattern. Let us start with the problem, let us read the problem statement.

(Refer Slide Time: 20:04)



SOFTWARE ENGINEERING

A Problem

In a graphical editor, by clicking on an object, one can obtain a copy of the original object. Obtaining a copy of an existing object is a common design problem in on-line compositions.

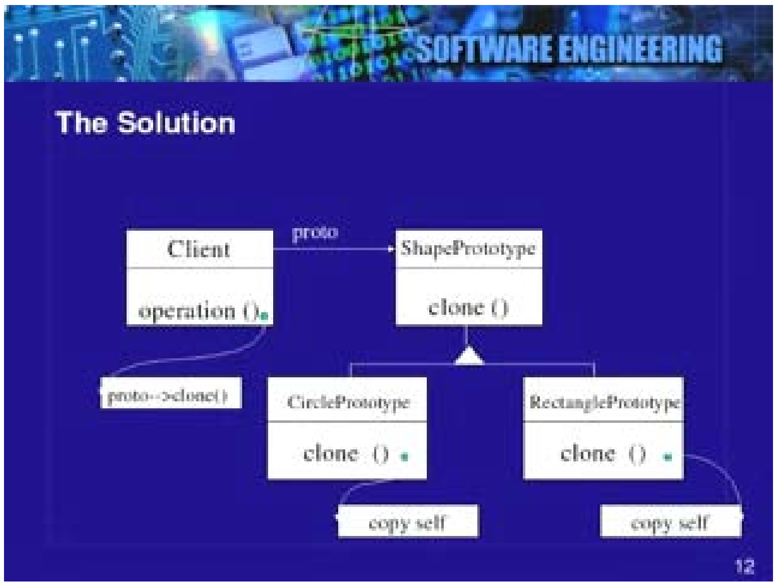
We can provide a design solution to solve this problem, and reuse this design whenever similar situation arises.

“In a graphical editor, by clicking on an object one can obtain a copy of the original object. Obtaining a copy of an existing object is the common design problem in online compositions. We can provide a design solution to solve this problem and reuse this design whenever similar situation arises.” All of you must have used one or the other graphical editor. You know that you can click on an object or a part of the figure and by right click you get small menu on the editor to copy the object and then you can use that copy. Let us say you want another copy of a rectangle, you can simply right click on it and then you get a copy of it.

You can apply this kind of technique to any shape on your drawing board. You can apply the same click and copy mechanism to lines, square, and to all shapes that you have drawn. That means a graphical editor is able to treat all these shapes as clone able one. In whichever application you have this requirement that the user have identified the object and the user want a copy of it from the object, the user is not going to specify the class of that object. The user just identifies the object and he can obtain a copy of that object. So whenever this requirement comes, the solution that you adopted in the graphical editor can also be used in that specific domain.

Now let us look at this solution. On this slide we see that you have organized your shape objects in a hierarchy. You have the super class which is titled as shape prototype. It is a prototype for all shapes and it supports a member function called clone. You have a circle prototype and you have a rectangle prototype. These are actually your circle class and rectangle class and they have to implement this method clone.

(Refer Slide Time: 22:10)

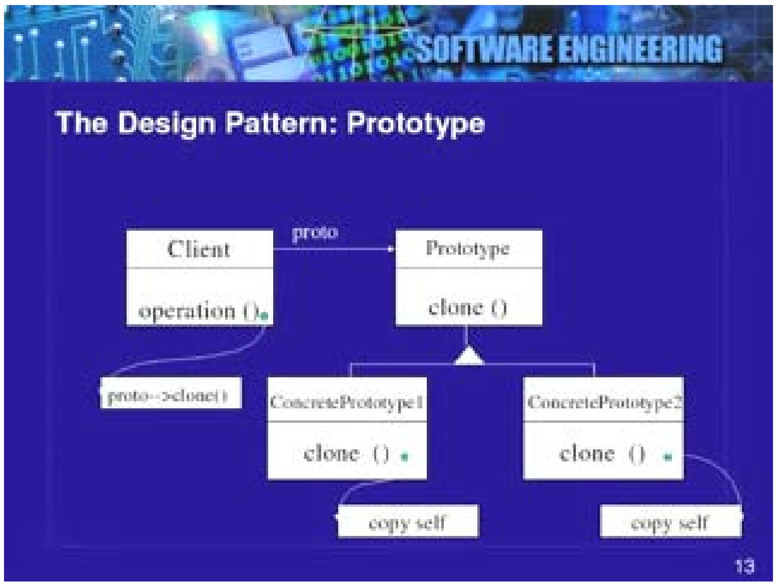


And there is a client code, which is the part of your application. It accesses all shapes and asks for clone of the selected shape. So client has this link into this ‘Shape Prototype’ hierarchy. Through this link from ‘Client’, through the polymorphism, you can point to any of these instances of the sub classes and ask for a clone of it. The client code does not need to be aware of the exact type of the object, whether it is a rectangle or whether it is a class when it is asking for a clone. So what happens here is that when you click on the mouse, the object handle is returned to you and on that object handle you simply invoke this clone operation and the clone operation gives you in return a clone of the object that you have selected.

The creation of a clone is done by the object itself. But the client code or the application code while asking for a clone is not aware of the type of object that the mouse click has selected. The mouse click just gives you a handle to the object that was selected by the mouse click and then once you have the handle to the object, that handle is of type say prototype, so treat it as an instance of type shape prototype. But the actual instance is from the subclasses. This is the solution that is adopted in by our shape hierarchy or our graph editor. These small boxes called ‘copy self’ show the implementation. How is the clone implemented?

You copy the ‘self’, so this is a pseudo code. When you are copying the object you know the concrete type. You will create an instance of the rectangle prototype and you will copy the entire state of this rectangle into the clone object and then return the cloned object. If you have sub objects in this object you have to copy them all. So it could be de-copy if needed. This is a specific solution in our graphical editor and the classes are named as shape prototype, circle prototype, rectangle prototype. But where is the pattern? This is concrete solution in a given application. How do we document and how do we represent the pattern so that it can also be applied in different domain? Here is the pattern description.

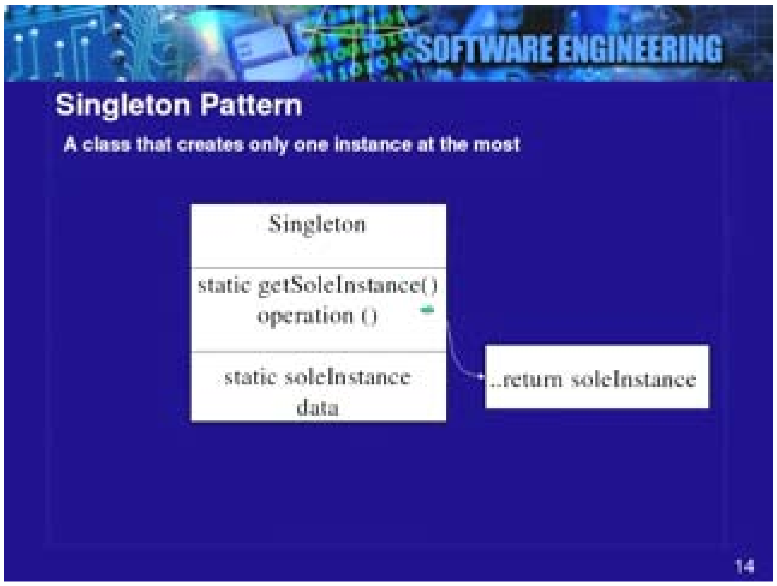
(Refer Slide Time: 22:10)



If you look at this slide, the shapes are gone and what remains is a prototype. You have super class as ‘Prototype’ and you have ‘ConcretePrototype1’ and ‘ConcretePrototype2’. There is a client and there is an operation. In the operation you are calling the clone on the prototype and if clone operations are implemented as copy by copying on themselves. This is your pattern and then it can be described in abstract using these abstract names. Any application that that has a similar need can identify its own concrete super classes and sub classes and map them on to this prototype hierarchy. So we can apply this prototype pattern in a different domain.

Now we are going to look at another creational pattern called singleton pattern. Singleton is a class that creates only one instance at most. Even if you try to create another instance, it should always give you a copy of instance that it has already created. So for example in this diagram a class is named as singleton. One question should come to your mind that if you just use the creational methods provided by your present programming language, will you not be able to create many instances of a given chart? Even if you want to make it as a single chart, take for example class stack, you can just say stack S1, S2, S3, S4 and you have got whole stack in C++ or you can say stack S1 is equal to new stack, stack S2 is equal to new stack and so on in Java when you got 4 stack.

(Refer Slide Time: 26:43)



SOFTWARE ENGINEERING

Singleton Pattern

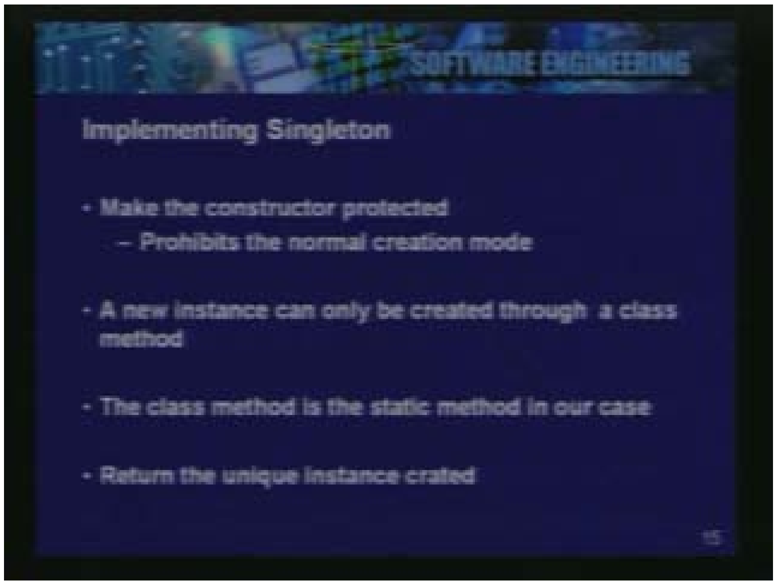
A class that creates only one instance at the most

14

How are you going to prohibit this kind of creation which the programming languages itself provides these? Solution here is that if you put your constructor in your private compartment, you first prohibit this kind of creation provides by the language and then support a specific method which is called as class method and not an instance method. Class methods are represented by keyword static in C++ and Java. So you can say here that you have a ‘getSoleInstance()’ method that is class method, static method. You have to call it on class, you may have to use operator, a different operator to call the method on the class directly. When you are getting an instance there is no other instance available you are getting your first instance. You have to invoke this method on the class and the getSoleInstance () method returns you with the sole instance which is also a class variable.

You can see that it is named as ‘static soleInstance’. The data and the operation belong to the instance are not static. This is your singleton pattern and the implementation that we just discussed. Let us make use of this facility of compartmentalization of private and public members. That is hiding your constructor by describing it as a private member. It prohibits the normal creation mode and then a new instance can only be created through a class method, which is described as static method. The class method then can be called by an environment to get the instance.

(Refer Slide Time: 29:24)


SOFTWARE ENGINEERING

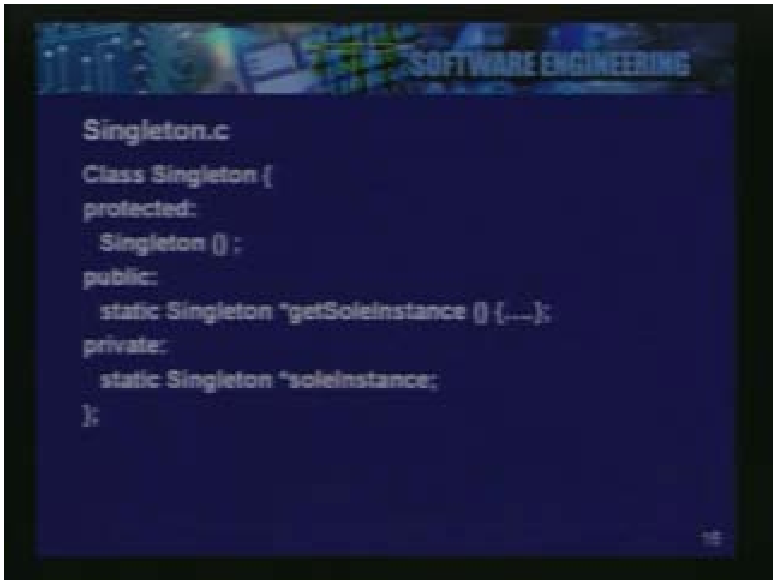
Implementing Singleton

- Make the constructor protected
  - Prohibits the normal creation mode
- A new instance can only be created through a class method
- The class method is the static method in our case
- Return the unique instance crated

15

And if you call it again and again the method can be programmed to check whether already an instance has been created or not. The instance is also a class variable. If the instance is already available, then it returns the same instance which is the unique instance in the class. Here is the pseudo code for typical singleton class.

(Refer Slide Time: 30:33)


SOFTWARE ENGINEERING

Singleton.c

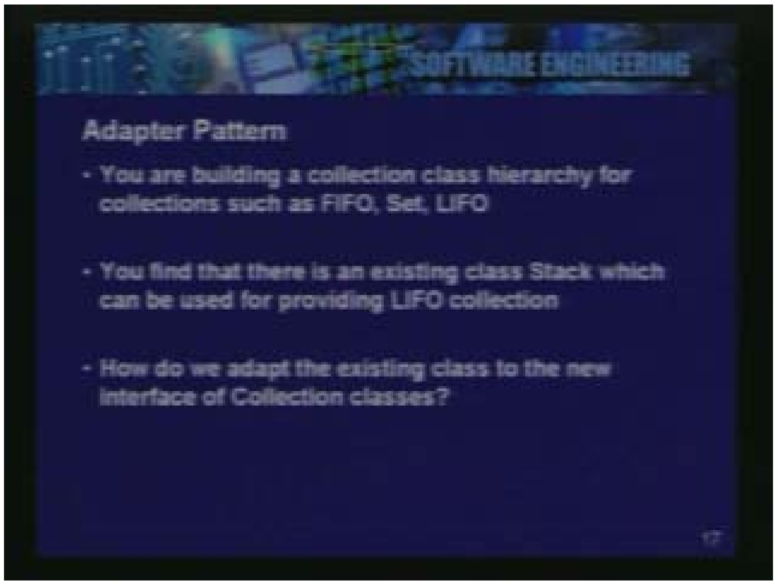
```
Class Singleton {
protected:
  Singleton () ;
public:
  static Singleton *getSoleInstance () {....};
private:
  static Singleton *soleInstance;
};
```

16

You have singleton that is a constructor and this class singleton which is protected. It is protected because your sub classes also should be able to benefit from pattern and then you have a static method called ‘soleInstance’ you have your singleton which is also ‘soleInstance’.

This ‘getSoleInstance’ method gives you this instance and the sole instance handles points to this sole instance.

(Refer Slide Time: 31:14)


SOFTWARE ENGINEERING

Adapter Pattern

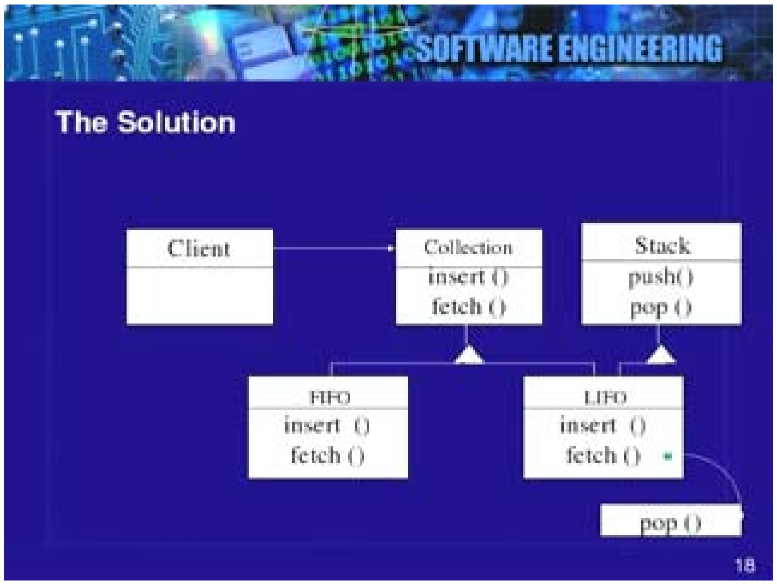
- You are building a collection class hierarchy for collections such as FIFO, Set, LIFO
- You find that there is an existing class Stack which can be used for providing LIFO collection
- How do we adapt the existing class to the new interface of Collection classes?

17

Now, we looked at two creational patterns. First one was prototype, second one was singleton pattern. Prototype allowed you to create object from a given object. Singleton pattern allowed you to create a unique instance of a given class. If you look at these requirements these are very generic, they can occur in any application and now you know how to go about designing for these requirements. Then the designs are also documented in terms of abstract name, classes, collaboration among them and the structure among them. Then you also have some concrete example documented along with the pattern. When you go through a given pattern dictionary, if you match the requirement with the intent provided along with the pattern descriptions, you can select a specific pattern and apply it in your application.

Now we are going to look at some examples of structural pattern. Let us look at the adapter pattern. Adapter is something like you are converting an AC volt to DC volt. This is an example from software domain. You are building a collection class hierarchy for collections such as FIFO, LIFO etc. FIFO is First In First Out collection set and LIFO is Last In First Out collection. This is a collection hierarchy and you may have an abstract class called collection and then all these are sub classes which you need to implement. Now, you find that there is an existing class called stack and it can be used for providing the LIFO collection. How do you adopt the existing class stack to the new interface of the collection classes? This below slide shows our solution. First let us look at the collection class hierarchy.

(Refer Slide Time: 33:33)

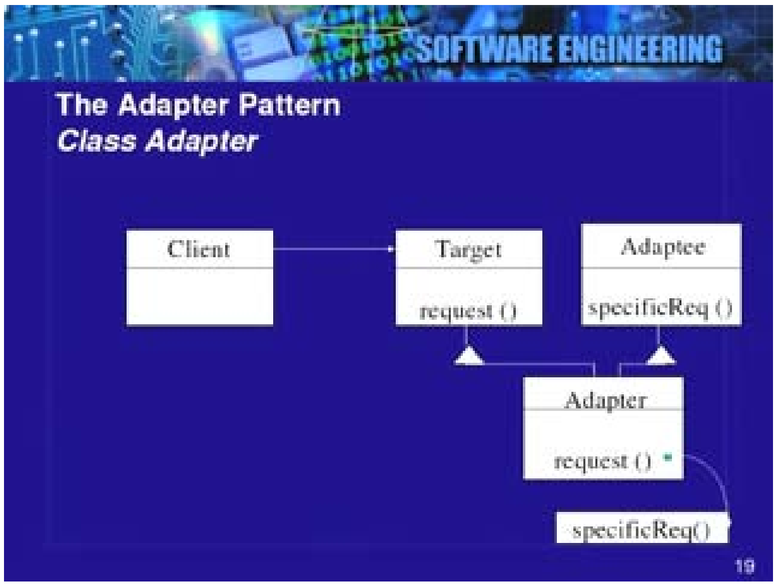


You have the ‘Collection’ super class which specifies two member functions called insert () and fetch(). You should be able to insert an item into collection and fetch the items out of the collection. Now, you want to build a LIFO type of collection and FIFO type of collection and these specific collection or these concrete collections are going to implement the abstract member functions declared in the super class or collection. When you are implementing LIFO, you want to use the existing class stack. But you see that it has a different interface on stack. You are calling it as push () or pop () and you want to map it to insert () and fetch (). How do you achieve this?

You are trying to now look at the intent of this pattern. We had described it earlier you are adopting this interface push() and pop() to insert() and fetch(), because the planned application here is going to use this object hierarchy to this specific interface only, that is insert and fetch. The client knows collection, which is of type ‘collection’. It might be either FIFO or LIFO and client application can simply call in certain fetch. But now since we already have something called stack with an interface called push and pop, why cannot we simply map push and pop onto this insert and fetch. That means that is to adopt one space to another. So the solution is adapter.

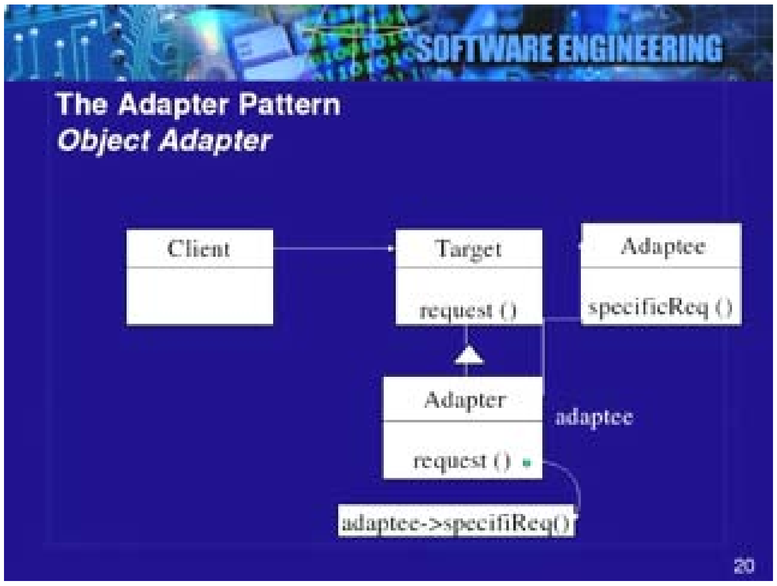
Now you must have figured out by this time, how we should be able to do that. In your LIFO class, you have insert and fetch. For example fetch implementation which has been given can be simply called pop now if you use multiple inheritance, LIFO inherits from this abstract class collection and a concrete class stack. And implements insert by a call to push and fetch by a call to pop. If you use inheritance, you can simply make these calls as if the functions are available on to you locally. The pattern can be described as follows. On the earlier slide we had the concrete solution, for an example that was collection hierarchy.

(Refer Slide Time: 36:15)



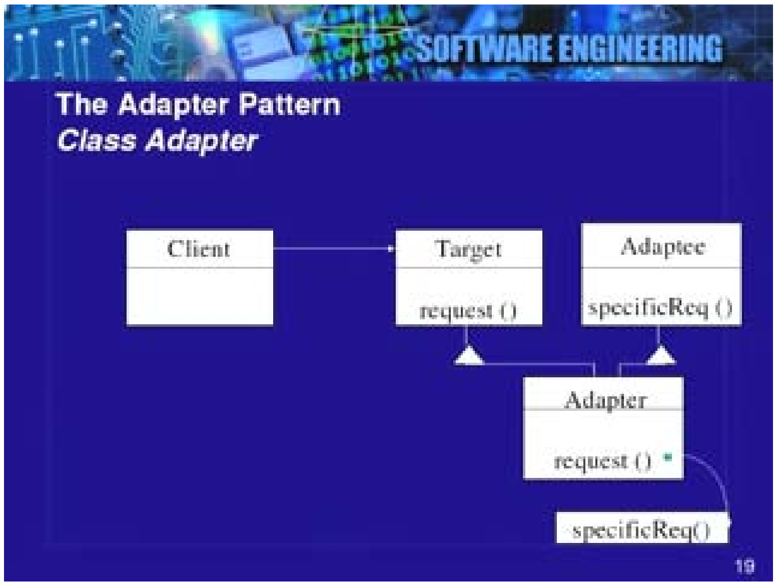
Here you replace your class collection by a more abstract name called target. You have ‘Target’, you have ‘Adaptee’ and there is the ‘Adapter’ which is the sub class. The adapter adapts the adaptee to the target. The client uses the ‘Target’ and the client knows only the ‘Target’ interface that is request. ‘Adaptee’ has a different interface called ‘specificReq ()’ and request can be implemented as call to ‘specificReq ()’, if you use multiple inheritance here. This is called class adapter. You are adopting the given class and to another class. So it is at class level. You can also have slightly different solution called object adapter. If you do not want to use multiple inheritance or if you want to change the objects to be adopted dynamically you could go for object adapter.

(Refer Slide Time: 37:14)



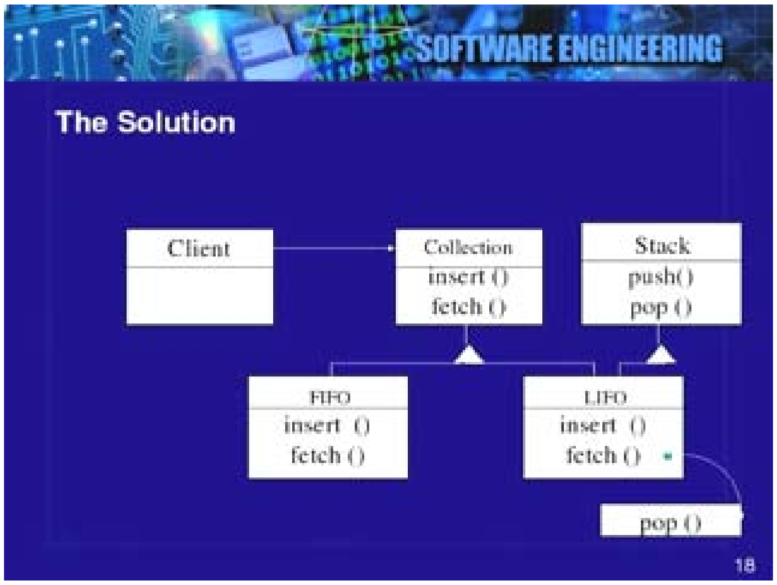
Instead of going for multiple inheritance, you use an instance directly. Now you can see that the request is implemented as adaptee arrow specific request (adaptee → specificReq ()). That means you are going to make a call on a given object you could use this object as a part in adapter. You could use an instance of the adaptee as the part in adapter. This is another solution or this is a different solution called object adapter. You can use object adapter if you do not want to use multiple inheritance. For example in this hierarchy if the target is partially implemented, then if you are using a language such as Java, if you have this as the full class, you wont be able to use class adapter because you have then two concrete classes.

(Refer Slide Time: 38:03)



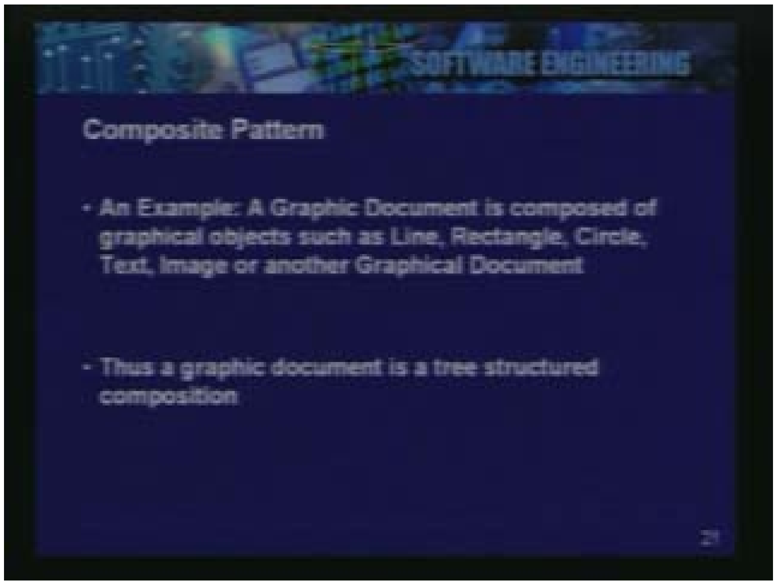
And you will have to go for a slightly different solution which is your object adapter or if you want to change the object dynamically then you cannot go for class level adaptation. If you go for class level adaptation, then you are stuck with one instance only, which directly becomes the part in the sub class. These are two solutions used for adaptation and these are the descriptions of the pattern. We saw that in the below slide which is a concrete example on ‘class collection’. We can describe the same pattern using abstractor. So we have a pattern now and whenever this adaptation requirement comes to us we can use this solution. Now we will go for another structural pattern called composite pattern. Here is an example: “A graphic document is composed of graphical object such as line, rectangle, circle, text, images or other graphical document.”

(Refer Slide Time: 38:42)


SOFTWARE ENGINEERING

The Solution

(Refer Slide Time: 39:11)


SOFTWARE ENGINEERING

Composite Pattern

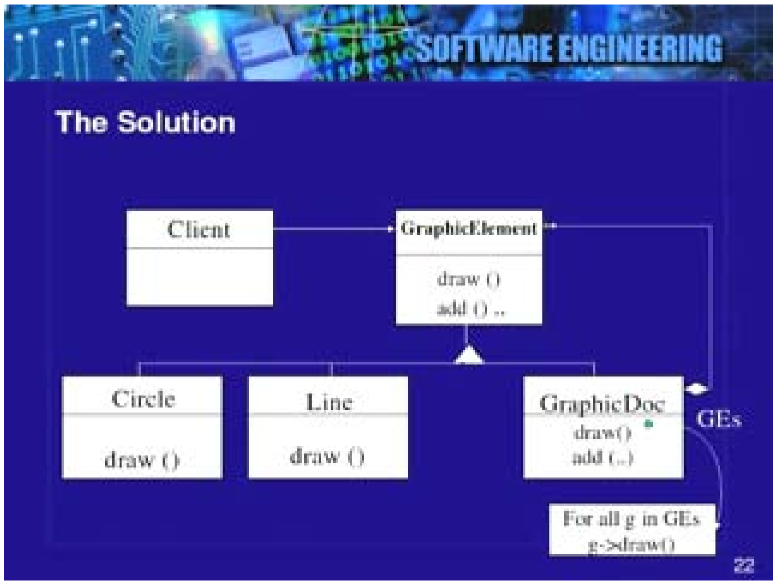
- An Example: A Graphic Document is composed of graphical objects such as Line, Rectangle, Circle, Text, Image or another Graphical Document
- Thus a graphic document is a tree structured composition

21

This says that the graphic documents inside a graphical document or say picture in a picture can also have shapes inside it. A picture can be composed of lines, squares, rectangles and so on. Or a picture can also contain picture. You want to represent a design for this. You could draw different object diagrams, say for example in one object diagram, you could have one picture at the root level and then it could have some leaf shapes directly and some two three pictures in it. Now those pictures can also have some leaf shape and one of the pictures in them and finally you will have leaf shape at the bottom of the tree. So you could construct such trees.

But then what should be the design? The design has to be closed, it must have finite number of classes. You cannot show it as a running tree. What is going to be your design for this particular requirement? Here is a solution. You represent this as composite. You can see that there are two interesting links amongst classes, one is inheritance and another is aggregation. Or we will call it composite.

(Refer Slide Time: 40:33)

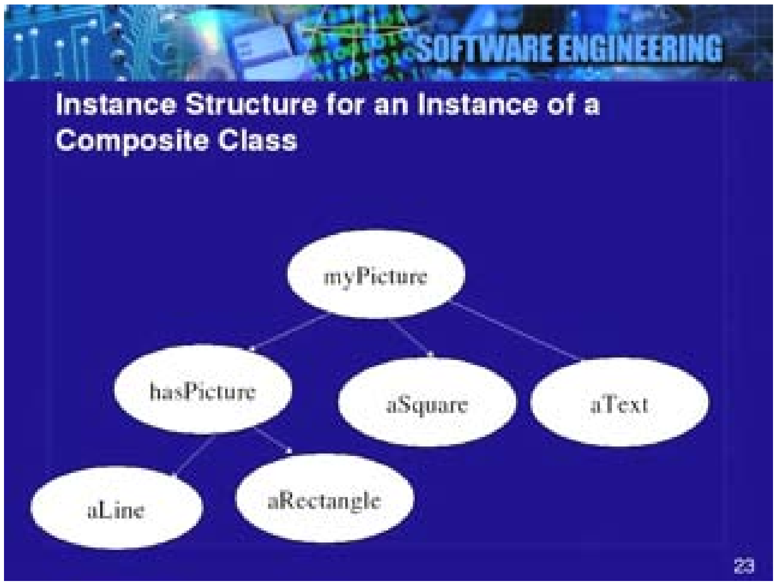


In UML there is slight difference between these two concepts. But now we will use them interchangeably. More generally you can call it as part-whole relation. So there is inheritance and there is also part-whole relation. You have graphical element as an abstract super class, which has functions like draw (), also another function called add () and then you have circle, line and graphical documents etc. You have various functions on graphical elements and you have different sub classes which implements graphical element. In this, GraphicDoc is a composite which is represented by a link from Graphic Element.

You have this part-whole relation and hence a graphical document can contain many graphical elements and through inheritance those graphical elements can be either circles, lines or GraphicDoc. This is how you can represent the design of your tree structured graphical document which contains leaf level classes and the intermediate graphical document which are composite. A client handles all of them as graphical elements. If you delete a graphical document it should pass on this delete to all the other components. For example, if you want to draw a given graphical element, if it happens to be a composite graphical element or a graphical doc, then the draw is implemented as for all g's in the link from GraphicDoc called draw. If the parts are leaf, then these draws (draw under 'Circle' or draw under 'Line') will be called and if the parts are composite the draw under 'GraphicDoc' will be called and they will be further propagated into the hierarchy.

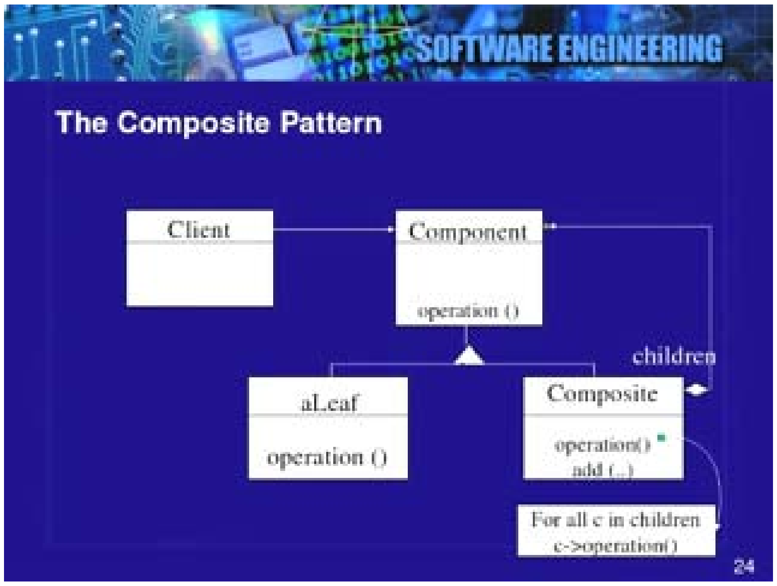
For example, this is one object diagram which you can construct after instantiating this given hierarchy.You have this ‘my Picture’ at this root level, it has this ‘has Picture’ which is basically another picture and the ‘has Picture’ has ‘a Line’ and ‘a Rectangle’.

(Refer Slide Time: 43:05)



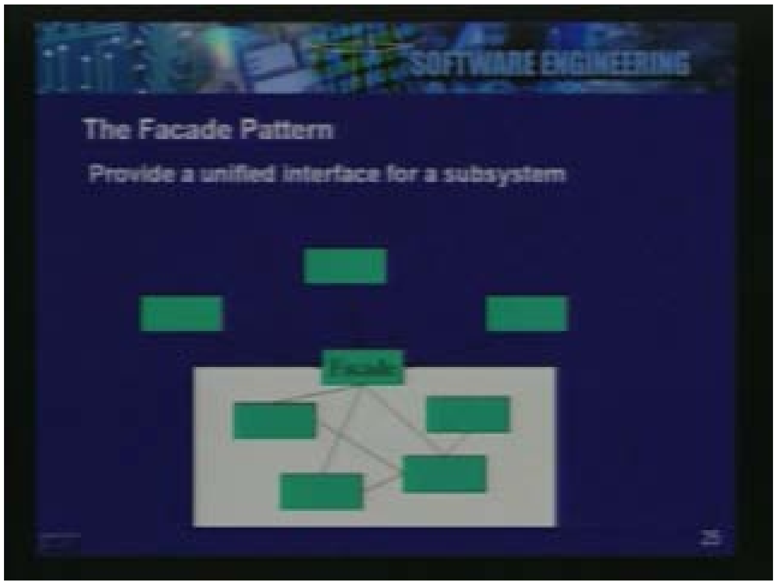
The ‘my Picture’ is composed of ‘has Picture’, that is one more picture inside it, ‘a Square’ and ‘a Text’. These three can be generated out of design that we have just seen. We have got a pattern or composite which allows us to represent such tree structure object diagrams through class designs or through designs expressed in terms of classes. Here is our composite pattern.

(Refer Slide Time: 43:38)



You have a component and then you have composite. You have a leaf and you could have many such leaves. Composite is also a component, but a composite has its parts which are also components. Now through this part-whole relation, the composites can contain a leaf as well as the composite. This is how you are able to represent this tree structure through a design in terms of classes and relationships. Hence the two relations are important, that is inheritance and part-whole. Let us look at another structural pattern which is called façade. It provides a unified interface for sub system.

(Refer Slide Time: 44:55)



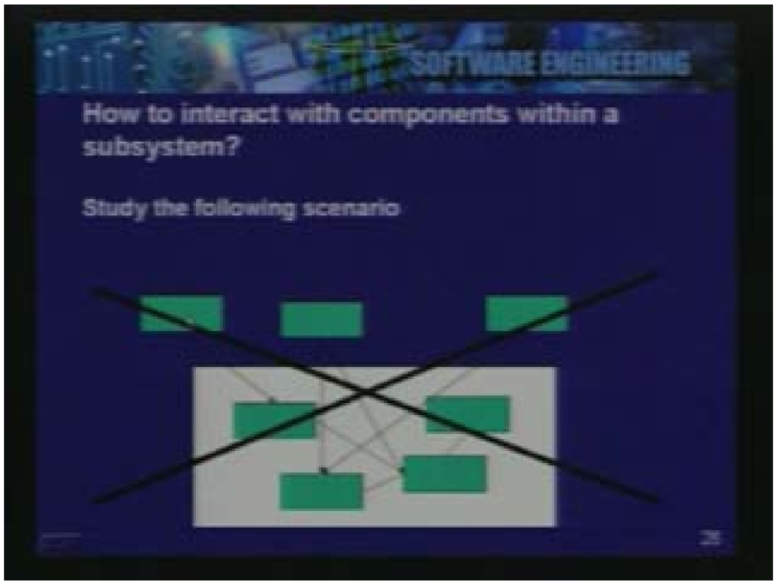
The Facade Pattern

Provide a unified interface for a subsystem

Look at the above picture. The white box is a sub system which has many classes and these classes are interconnected and collaborate. These outer green boxes are outside classes, which are outside this sub system described by the white box. All of them are interacting with this sub subsystem through one class. This class is an interface to the entire sub system and this class is called façade. Now, compare this solution with the below solution. We have rejected it because the external classes are interacting with the sub system directly through different classes in the sub system.

For example, the middle component which is outside the subsystem is interacting with two classes inside the subsystem and so on. If there is any change here, it is very hard to reflect the change in entire system as there are too many dependencies. The separation of concerns is violated. The sub systems must be nicely encapsulated and behind sub system interface which is called façade. This is one structuring pattern and you should structure your sub system in this way. Let us look at another structural pattern called proxy pattern.

(Refer Slide Time: 45:31)

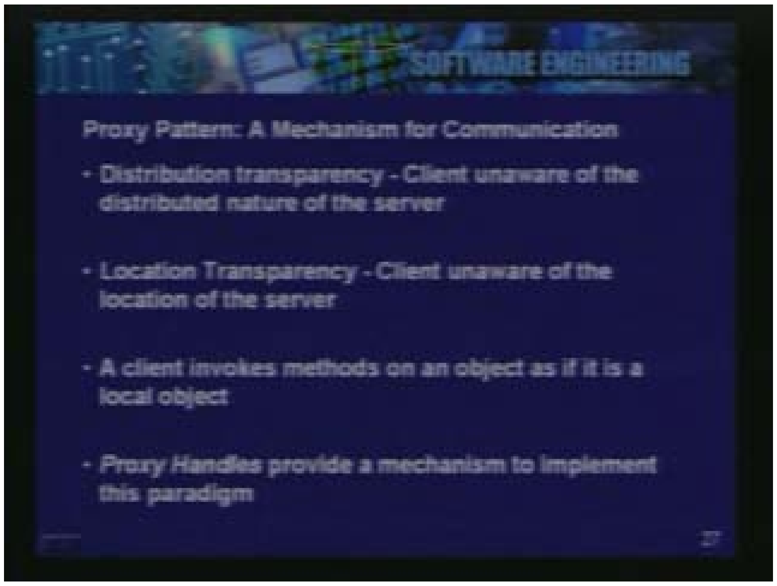


How to interact with components within a subsystem?

Study the following scenario

It is a mechanism for communication. You imagine a distributed system scenario in which an object on one machine wants to interact with an object on another machine, can we make the distribution transparent?

(Refer Slide Time: 46:27)


SOFTWARE ENGINEERING

Proxy Pattern: A Mechanism for Communication

- Distribution transparency - Client unaware of the distributed nature of the server
- Location Transparency - Client unaware of the location of the server
- A client invokes methods on an object as if it is a local object
- *Proxy Handles* provide a mechanism to implement this paradigm

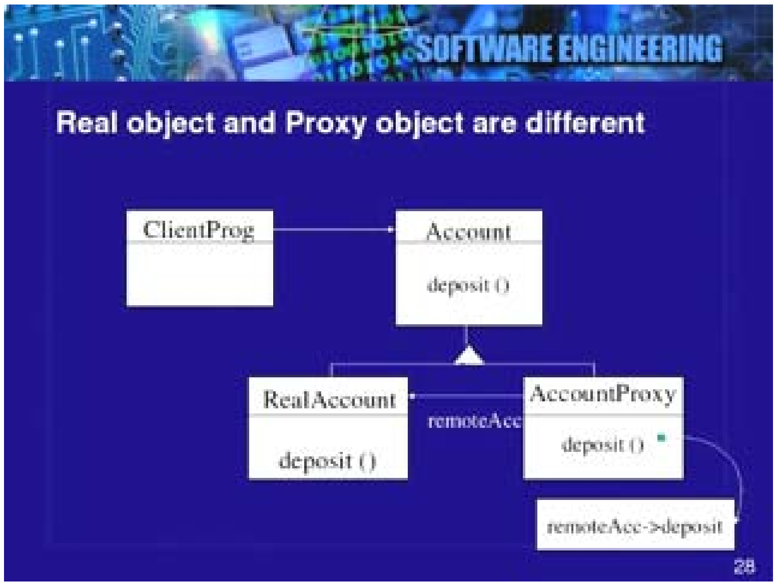
27

Can the object that wants to make a call on the remote object be made unaware of this kind of distribution and networking? If this local object can have an interface, a local interface to the remote object will simply invoke member function on the local interface. It will not be even aware that this local interface, the implementation of local interface is contacting the remote object over the network.

So the definition of distribution transparency: The client is unaware of distributed nature of the server. In the sense the server is on a different system and the client is on a different system on a different machine. Location transparency: The client is unaware of the location. Firstly that it is separately located in a distributed system and secondly it does not even need to know where it is located. A client invokes a method on object as if it is its local object. And that local object which represents the remote object in local environment is called proxy. So proxy handles provide mechanism to implement this communication paradigm.

So here is a solution. For example, you have an account object which is remote object and you want to make a transaction on the account object. The client program can be given an instance of this 'Account Proxy' which implements the interface provided by the account object, that is, deposit(). The implementation is given in the small box. You are calling deposit on the remote actual object which is remote account (remote Acc).

(Refer Slide Time: 48:18)

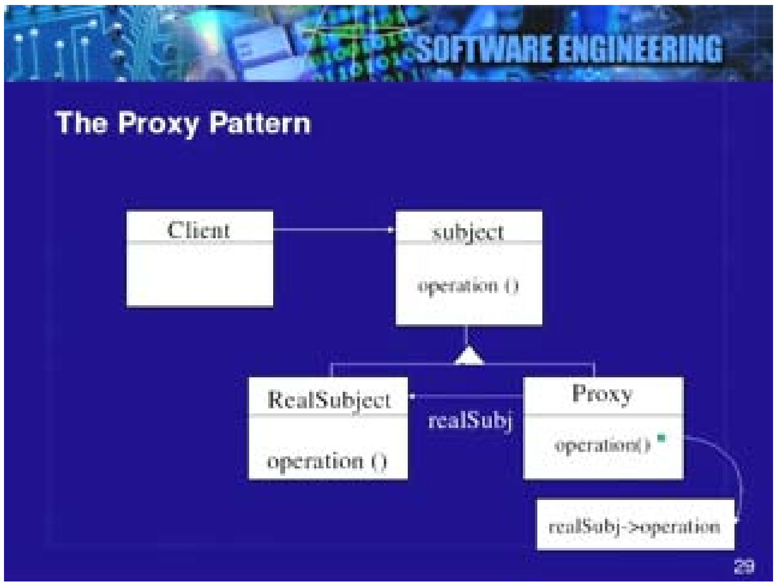


SOFTWARE ENGINEERING

Real object and Proxy object are different

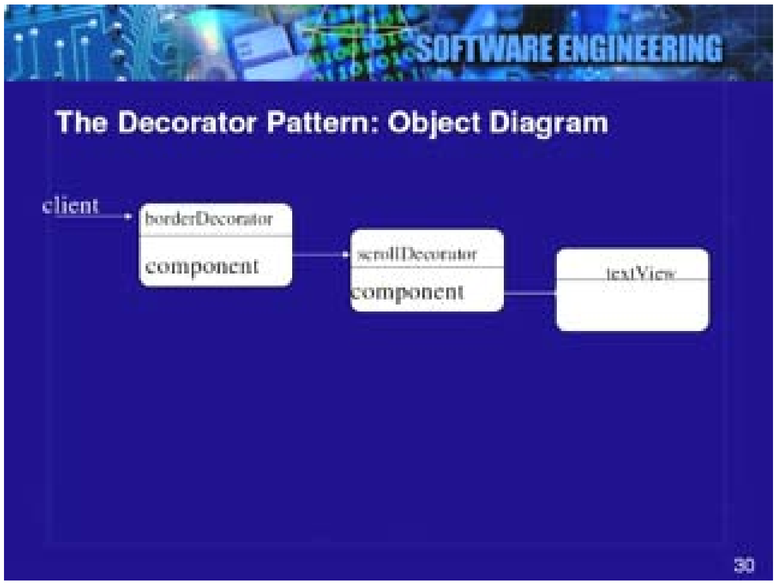
This is a 'Real Account' which also implements the account interface. That is deposit and 'Account Proxy' has a link to the remote account. This could be over the network. So the client does not distinguish or client is unable to distinguish between the real and the proxy because both of them look the same. Both of them follow the super class which is common. 'Real Account' implements it actually. The actual deposits will be implemented here balance will be maintained in the real account A proxy simply sends the call to remote object and results are returned to the client. So this is an interesting solution for a distributed scenario, in a distributed object system which is called proxy. Proxy pattern is represented in this slide.

(Refer Slide Time: 49:37)



You have a subject and you have a ‘Real Subject’ and a ‘Proxy’. The ‘Real Subject’ actually implements the subject and the ‘Proxy’ simply roots the call to a ‘Real Subject’. Let us look at another structural pattern called decorator. Look at this object diagram; a client is making a call and the call is going through different objects and finally reaching its server.

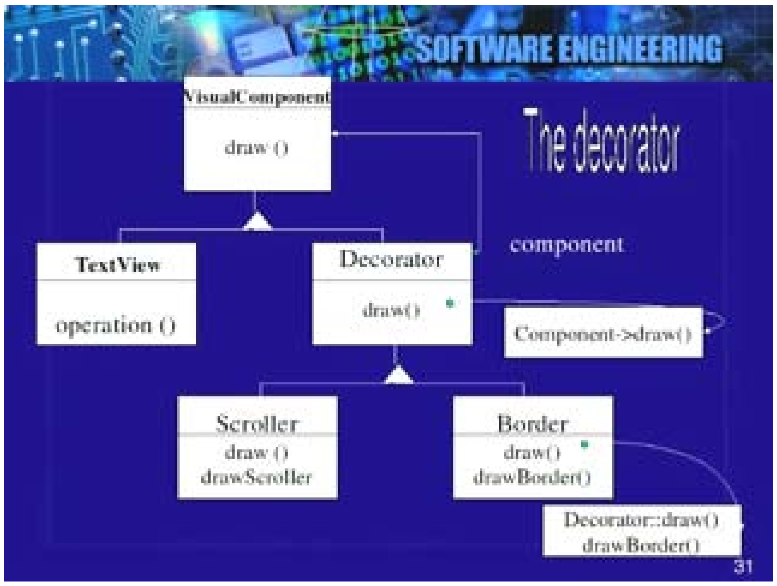
(Refer Slide Time: 49:55)



The ‘text View’ has a scroll Decorator and a border Decorator. So you have different kinds of borders and view. Now you want to change the border, you want to add different kinds of decorators on the text view. You should be able to add these decorators.

On a given object you are supporting a member function. When you invoke a member function on that object, you want to invoke some thing else or something extra in addition to what the object provides. For example you have a shape and you want to draw border from the shape or change those borders. In this case you have a text view and additional functionality that you want to support is trying different borders or scroll bars and so on. Can this be done without the client knowing about what you are going to add in between and how many such decorators are you going to add? We are calling it decorator because it is that the pattern suits very well for user interfaces. But same requirement can also occur in a different application where you want to add something new on top of what is existing and you want a client not be aware of this internal addition.

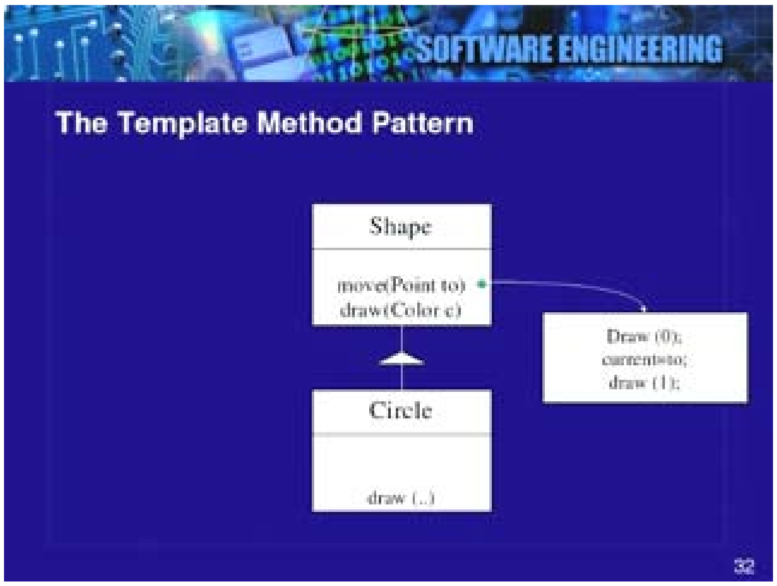
(Refer Slide Time: 51:30)



This is your hierarchy and we are not going to go through all details of this particular structure. But you can study the structure and the implementations through the reference book that I am going to share at this end of this lecture. But what you can notice is that, again you have two kinds of relations. You have one link to the super class and you also have the inheritance. You can see that there are different kinds of decorators and the decorator has the link to actual component. The 'Decorator' and the actual component or the 'Text View' exactly supports the same interface. Hence the client is aware of exact type of the object which it is talking to, whether it is talking of the actual server or the decorator. Let us look at one example of behavioral pattern which is template method.

This is again a familiar example where you have a class called shape and when you want to implement move operation on shape, it can be implemented as draw existing shape with color zero that is black it out, then change location and then draw the shape again. This 'move' method has been implemented in the super class. But that 'draw' method has to be implemented by the actual shape.

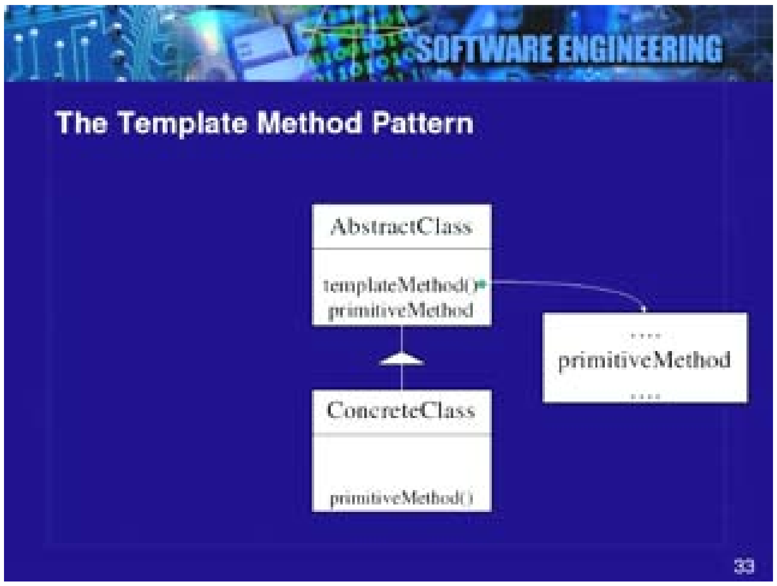
(Refer Slide Time: 52:30)



The Template Method Pattern

We have an implementation of ‘move’ which is in terms of abstract method which is not implemented in the shape. The method move is called template method, where as method draw is called hook method in this pattern. This is a very generic requirement which occurs in different applications where you must be able to implement such template method. The slide below shows the abstract description.

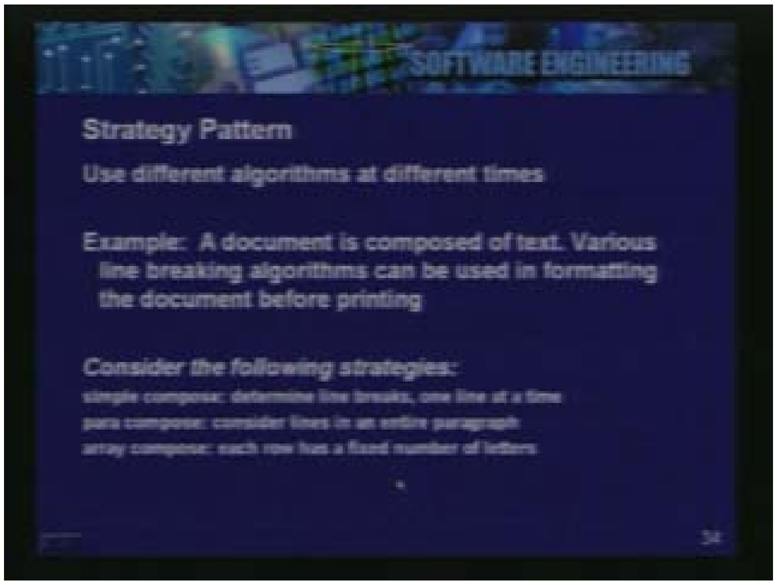
(Refer Slide Time: 53:33)



The Template Method Pattern

You have the template method which is implemented in some source primitive method. The primitive method is what is implemented in concrete class. Now we will quickly look at another pattern called strategy pattern you should be able to use different algorithms at different time, different strategies at different time.

(Refer Slide Time: 53:46)



SOFTWARE ENGINEERING

Strategy Pattern

Use different algorithms at different times

Example: A document is composed of text. Various line breaking algorithms can be used in formatting the document before printing

Consider the following strategies:

simple compose: determine line breaks, one line at a time

para compose: consider lines in an entire paragraph

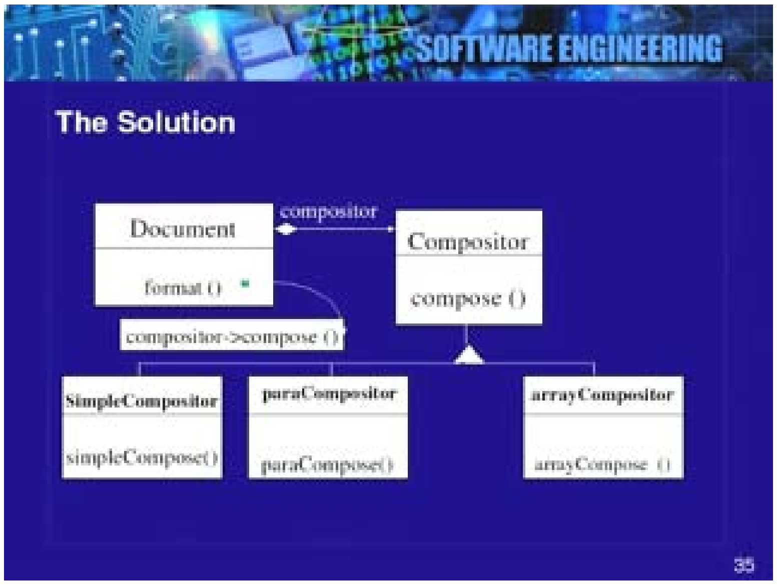
array compose: each row has a fixed number of letters

34

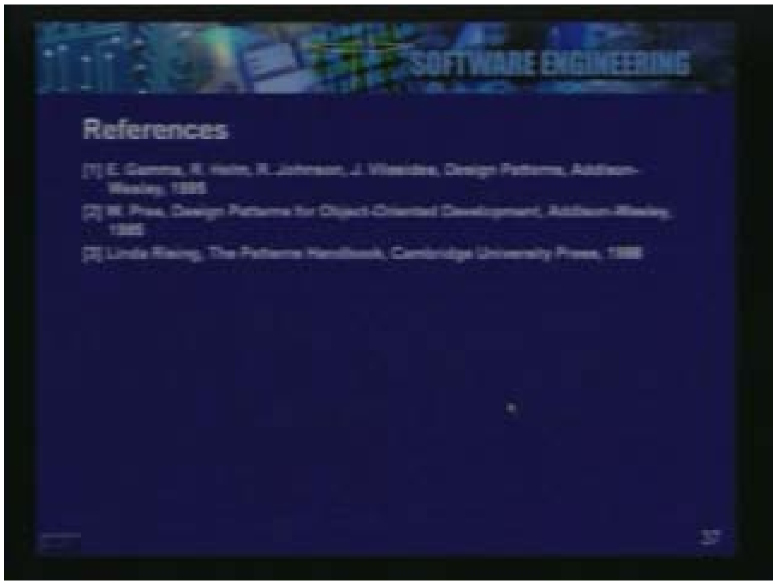
For example: A document is composed of text and various line breaking algorithms can be used in formatting the document before printing. You could have simple compose; determine line breaks, one line at a time or you consider lines in an entire paragraph or each row could have fixed number of letters which is array type of composition. These different strategies you can use for justifying your text document and this is your solution.

You have a document and you have your sub class, you have various sub classes of your compositor. The document contains the compositor and the compose is implemented in different sub classes. This is your strategy pattern and you have many strategies concrete strategies. All of them implement the algorithm differently. Finally we will look at these references to this lecture. The first reference is Design patterns by Erich Gamma, Helen, Johnson, Vlissides, which is published by Addison Wesley.

(Refer Slide Time: 54:30)



(Refer Slide Time: 54:51)


SOFTWARE ENGINEERING

References

[1] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns, Addison-Wesley, 1995

[2] W. Pree, Design Patterns for Object-Oriented Development, Addison-Wesley, 1995

[3] Linda Rising, The Patterns Handbook, Cambridge University Press, 1998

37

You have another book called Design patterns for Object Oriented Development by Wolfgang Pree (W. Pree). And there is a nice reading available, it is called pattern handbook by Linda Rising. There are many papers also available on this subject. You go through all these materials in order to get more explanation on the slides that I have described.

Thank you