

FUNDAMENTALS OF OBJECT ORIENTED PROGRAMMING

Lecture50

Lecture 50: Structural Patterns

Thank you. Welcome to Lecture 50: Design Patterns. So, in today's lecture, let us talk about structural patterns, right? So, the structural patterns we are going to see are the adapter pattern, decorator pattern, composite pattern, and the proxy pattern. So, they are all defined as how objects and classes can be combined to form larger structures, right?

So, in fact, these are all ensuring that the system is flexible and easy to maintain. These patterns focus on simplifying relationships between entities and ensuring modularity and reusability, right? So these are all the points that explain why structural patterns are so important. Structural patterns enhance modularity. So, this provides flexibility, and these structural patterns enable the reuse of components in different contexts. So, we call it code reusability. We have scalability, and we have improved readability, right? So, these are all the reasons why we can use structural patterns, all right?

So, under structural patterns, the first one we are going to see is the adapter pattern. Right, so the main idea of using the adapter pattern is to convert the interface of a class into another interface clients expect. The adapter pattern enables two incompatible interfaces to work together. Let us take an example, such as connecting a legacy system with a modern API. All right, so let us consider the adapter interface by considering the class Target, right? So, which is the abstract class? In fact, you have the pure virtual function, all right? The pure virtual function is the function called Request, right? As I already said, it is a pure virtual function, Request, all right? So that means the Target class has to be inherited further, meaning we have to define this Request So, let us have another class, the Adaptee class, called Legacy System, right? So, under Legacy System, you have the function called Specific Request, which has one cout statement called Legacy Request, right?

Adapter Pattern: Overview

✎ **Purpose:** Convert the interface of a class into another interface clients expect.

✎ Enables two incompatible interfaces to work together.

✎ **Example:**

► Connecting a legacy system with a modern API.

```
1 // Adapter Interface
2 class Target {
3 public:
4     virtual void request() = 0;
5 };
6
7 // Adaptee
8 class LegacySystem {
9 public:
10     void specificRequest() { std::cout << "Legacy Request\n"; }
11 };
```



cout legacy request. So you have the adaptee class. The name of the class is legacy system. So now we have adapter class, right? So which is inheriting the abstract class called target, right?

So, this is abstract class over here. I have one pointer object adapty under legacy system. So, I have defined legacy system class over here and then this is a private pointer object adapty and then we have a constructor adapter. So, it is a one argument constructor. So, legacy is a pointer object and you have the class legacy system.

right and legacy is copied into adapty over here in this constructor all right. And we have one function in fact we have to define this request function right because it has to be overridden we are defining this in the derived class all right. So that means this will be overridden the previous one will be overridden and here we the adapty, right, the object adapty. So, legacy is a pointer, yes, that means this adapty, the object which is info, in fact, it is a pointer object, right, which is invoking specific request function, right, specific request function.

member function right so specific request member function is over here so that means the legacy request will be printed right if at all it is called so that we will see in the main program so your adapter class is over here so let us consider the main So, in the main program assume that you have an object called legacy. So, the legacy is under legacy system which is the class and you have an object adapter all right. So, that means you are pausing the reference of legacy all right. So, that means adapter constructor will be called line number 16.

Adapter Pattern

```

12 // Adapter
13 class Adapter : public Target {
14     LegacySystem* adaptee;
15 public:
16     Adapter(LegacySystem* legacy) : adaptee(legacy) {}
17     void request() override { adaptee->specificRequest(); }
18 };
19
20 // Client
21 int main() {
22     LegacySystem legacy;
23     Adapter adapter(&legacy);
24     adapter.request();
25     return 0;
26 }

```

So, now your legacy is nothing but the address of legacy all right. So, the legacy is a pointer object. So, legacy and address of legacy is point address of legacy in the main and the legacy over here in the constructor is pointing to the same location. Legacy and address legacy is pointing to the same location right. So, whatever we are passing that will be copied into adapt.

So, now this adapter object is invoking request right. So, the request is a overridden function. So, adapte will invoke right. So, the adapte will be invoke this specific request right. This will invoke the specific request.

So, a specific request over here. So, the legacy request will be printed, right? So, when it is invoking, the legacy request will be printed, and line number 10 will be printed. Right, return 0. So, when I run this code, as I said.

So, the legacy request will be printed. So, this is what happens in the case of the adapter pattern, and the next pattern is the decorator pattern, right? So, the main idea of this decorator pattern is adding scroll bars to a window without modifying the window class, all right. The purpose of using the decorator pattern is adding new responsibilities to objects dynamically. It also provides a flexible alternative to subclassing for extending functionality.


So, here you go. So, a slightly bigger program. So, let us consider the C++ program. You have a class UI component, right? And you have a virtual destructor is equal to default, right?

And you have the virtual. So that means this is the abstract class UI component. So you have the function called render, which is a pure virtual function. All right. So let us see what further it is doing.

So you have class window, which is inheriting UI component. So that means render has to be defined over here. All right. So it is returning window. All right.

So it is overridden. So this should be overridden. So it is returning window. So, it is returning window. You have a class decorator over here, which is also publicly inheriting UI component.

```
19 // Abstract Decorator Class
20 ~ class Decorator : public UIComponent {
21     protected:
22         UIComponent* component; // Wrapped component
23
24     public:
25         Decorator(UIComponent* comp) : component(comp) {}
26         virtual ~Decorator() { delete component; }
27
28 ~     std::string render() const override {
29         return component->render(); // Delegate render to the wrapped component
30     }
31 };
32
```



You have protected pointer object component under UI component. And you have a public, right, it is in fact the constructor, decorator constructor. You have a pointer object under the class UI component, right. So, COMP, whatever you are passing will be copied into component. That is the meaning.

And then here we have the destructor decorator. So, which is deallocating the component all right. So, here we have the render which will be overridden right. So, we already defined in the case of class window. So, here we are defining for decorator.

So, render. So, here you have overwrite all right render yeah overridden. So, the component is invoking render all right. So, whereas here return window. So, here the component is invoking render anyway the detail explanation is given in the common statement.

So, delegate render to the wrapped component. So, the decorator class is over at line number 31. So, now you have another class called scroll bar decorator, all right. So, which is inheriting decorator in a public access modifier, right? Scroll bar decorator, decorator, scroll bar decorator, right? So here you have UI component, right? Then you have decorator, then you have scroll bar decorator, all right. So in this case, when I am talking about the scroll bar decorator here,

you have the constructor scroll bar decorator, right? Comp is a pointer object under UI component

right. So, UI component is over here, right. So, UI component is over here, the base class or you call it as abstract class, right. And the comp is copied into decorator, right. That is what is happening in this constructor.


And now again, you have render overriding function. So, now it will return. So, that means the component is invoking render, right. So, plus you are going to get this particular output, right, with scroll bar. So, this is called the scroll bar decorator, previously decorator, and the base class is UI component, multi-level decorator, and scroll bar decorator, correct.

So, yeah UI component and it is decorator. So, again you have border decorator all right. So, which is inheriting decorator and you have a constructor. So, constructor you have an object comp under the class UI component the pointer object right. So, comp is copied into decorator.

Again same you have a render function which is a overridden function. Component is invoking render. Here you are getting with border right. Class is over at line number 51. Border decorator is over at line number 51 and scroll bar decorator is over at line number 41.

So now let us go to the main program. So the main program you have the pointer object simple window right. The pointer object simple window that will be upcasted to window right. Simple window is a pointer object. correct under ui component it is upcasting to window right so window we don't have any constructor fine right now and you have right so here what will happen suppose if it is calling this render

```
53 - int main() {
54     // Step 1: Create a simple window
55     UIComponent* simpleWindow = new Window();
56
57     // Step 2: Add a scrollbar to the window
58     UIComponent* windowWithScrollbar = new ScrollbarDecorator(simpleWindow);
59
60     // Step 3: Add a border to the window with scrollbar
61     UIComponent* decoratedWindow = new BorderDecorator(windowWithScrollbar);
62
63     // Step 4: Render the final decorated window
64     std::cout << decoratedWindow->render() << std::endl;
65
66     // Clean up memory
67     delete decoratedWindow;
68
69     return 0;
70 }
```



window will be returned all right. So, suppose it is calling right. So, let us call this function later right. In fact, we are calling it line number 64. Now, you have another object called window with scroll bar that is a pointer object under UI component upcasting with a scroll bar decorator by pausing this simple window all right.

So, you are making the connection all right that upside class then the subclass In fact, we are having two subclasses and then, right, you are having all the inheritance, all right, the hybrid inheritance, I mean to say, all right. So, you are passing simple window. Simple window is the pointer object under UI component, pointer object under UI component. And then I have decorated window, right, the pointer object under UI component, right, where this will be upcasting with border decorator.

So, border decorator you are pausing window with scroll bar. So, one by one. So, you can see in the common statement. So, initially you are creating a simple window right. Then you are adding with a scroll bar to the window.

So, one by one suppose if I am calling this render yes I am going to do it in line number 64. So, till upset class you do not have the definition and after you have any inheritance right because one by one you are doing. If you look at the decorated window upcasting with the border decorator. And then you are pausing window with scroll bar. This you are pausing.

That is why I am putting arrow. One by one I am going. So line number 64, when the decorated window, so this one, this object is invoking render. So let us see. It will be the interesting output.

So, decorated window when it is invoking render. So, that means it will be upcasted to border decorator. So, first I go to border decorator. So, border decorator here you have component. So, which is in fact whatever will be copied.

So, in fact we are passing window with scroll bar. So, this will be copied into comp. So, in fact the comp will be copied into decorator. right and then this component correct so which is in fact available over here in the ui component so the component will invoke render right so when it is invoking render so one by one so go above the arrow all right go above the arrow so when it is invoking this right since you have the scroll bar decorator correct so scroll bar decorator so one by one it is going all right So this is you are upcasting.

So it goes over here. So component render with scroll bar, right. So component render. So when it is calling, it will go to the symbol window, right, which is upcasted by the window. So what is there in window?

Window, it is returning window. So that means first window will be printed, right. So there is no break. I mean to say there is no new line. Window will be printed first, right.

So, in this case, the window will be printed. So, that means when it is calling a window with a scroll bar. So, here in this case, when it is calling the total window with a scroll bar, all right. When it comes over here, a window with a scroll bar and a border will be printed, right? So, this is exactly what your line numbers 55, 58, and 61 say, right?

So, the window is the first output returned, right? So, that will be printed. So, when it is printing a window with a scroll bar, right? And then a window with a scroll bar for this. Plus, with a border. So, a window with a scroll bar and a border will be printed.

So, you can see the output. Yes, this is what I said, all right. So, this is how the decorator pattern works, all right? I mean, this is an example of a decorator pattern. So, let us consider the program, right? So, here, as usual, we have the abstract class FileSystem, right? Which is a Java code, right? Which is a Java code. So, this abstract class has an abstract method called display.

Now, right, so you have a file system, right, which is the abstract class. So that means this display has to be defined in the subclass, right. So we have a file,

class file. So under class file, you have this display. So here you go, line number 10.

So now what is happening? So you can see over here, the file is extending file system. You have one data member called name under string. Under the data type string, and here you have the constructor file, one argument constructor name string, and name is copied into this dot name. And here you have the display, which is defined system dot out dot println and name will be printed, OK? And here you have folder again. In this folder, you have display, alright? So this is also having display. And what is the name of the class?

Folder. Composite class called folder. Alright. So, this is also having display at line number 16. Yeah.

Composite Pattern: Overview

- Purpose:** Compose objects into tree structures to represent part-whole hierarchies.
- Allows** clients to treat individual objects and compositions uniformly.
- Example:**
 - File systems with folders and files.

```
1 // Component
2 abstract class FileSystem {
3     abstract void display();
4 }
5
6 // Leaf
7 class File extends FileSystem {
8     private String name;
9     public File(String name) { this.name = name; }
10    public void display() { System.out.println(name); }
11 }
```

Handwritten notes: "Java" with an arrow pointing to the code, and "10 → display()" with an arrow pointing to line 10 of the code.

Diagram: A box labeled "FileSystem" has two arrows pointing up to it from boxes labeled "File" and "Folder".

Video inset: A man in a white shirt speaking.

Interesting, right. So, here you have the abstract display. File system has having abstract display. So, now the folder is extending file system. So, the file system has the object children, right.

So, object children under the class list of file system, right. So, here you can see the memory is allocated with the constructor array list. Memory is allocated with the constructor array list. And line number 15, we have add method. right, so we have the method call add, whose return type is void, here I have an object child under file system, right, so here the children, the object under list is invoking add, right, by pausing child, invoking add by pausing child and the display, it is a range for loop, that means the children, right, so that you are pausing will be copied into, that means both will point to the same location,

Or same identity as map. Children. Right. And child. The child is invoking display.

Right. And the child is invoking display. So here you have. So it will try to. Right.

The display the content. In the file system. Right. But unfortunately, it is an abstract method. So therefore, this will be invoked.

Right. So that we will see when you are calling the main. Right. So the main you have root object. So, the root object under folder and your constructor is also folder. Fine, well and good. It will be there only whenever you are calling any method, right?

The root is calling the method add right. So, add you look over here right you look over here it is a constructor file right. So, file of you are pausing file 1 dot txt and file you have the constructor you are pausing the text one right file one dot text so file one dot text that you have passed so this dot name will be file one dot text right and here you have add right the function is add so that means file one dot text will be added into the list right so this will be added into the list right so this is what exactly happening in fact we had seen several times how you are adding right the observer or right so many examples that we had seen so this will be adding right add again root is calling add file to dot text now so the same explanation next in the list is file to dot text right so now root is invoking display

So root is under folder invoking the display, right? So your constructor is also folder. So therefore, this display will be invoked and you have range for loop, right? So children that will be copied into child. Anyway, these things, right?

So we have over here. So now child dot display, right? So child is invoking display, right? So, that means in the superclass, I mean to say the abstract class you have display. But unfortunately, this is being defined in the subclass file.

Now, it is accessing line number 10. So, `system.out.println` name. What is the name? File 1 dot text. So that will be printed first.

Range for loop. Again child dot display is invoked. Line number 10 again. So now file 2 dot text will be printed. Right.

Composite Pattern: Overview

- Purpose:** Compose objects into tree structures to represent part-whole hierarchies.
- Allows** clients to treat individual objects and compositions uniformly.
- Example:**
 - File systems with folders and files.

```

1 // Component
2 abstract class FileSystem {
3     abstract void display();
4 }
5
6 // Leaf
7 class File extends FileSystem {
8     private String name;
9     public File(String name) { this.name = name; }
10    public void display() { System.out.println(name); }
11 }
  
```

Handwritten notes and diagram:

- A diagram shows a hierarchy: **FileSystem** (root) contains **File** and **Folder**. Arrows point from **File** and **Folder** to **FileSystem**. The word "root" is written next to the arrow pointing to **FileSystem**.
- Handwritten text: "10 → display()", "display()", and "16" with arrows pointing to the **display()** method in the **File** class and the **display()** method in the **Folder** class.
- A small video inset shows a person speaking.

So this is what exactly happening. The output is file 1 dot text and file 2 dot text. Right. I hope it is clear for everyone. So now the next pattern is the proxy pattern.

What do you mean by proxy pattern? So, when you are loading large images lazily in a graphical application, many image processing applications, all right. So, this proxy pattern, all right. So, which is the main idea of using this proxy pattern, which is used to provide the surrogate or placeholder for another object. And more interestingly, this controls access to the original object.

So, let us take the example of a image let us have a class image all right and this is a in C++ right. So, you have the virtual function pure virtual function display which is equal to 0 that means I have to define later and real object all right. So, let us say class real image which is publicly inheriting image. Here you have file name whose data type is string right you have public constructor let us say real image file under string right. So, file is an object and the string is the class right or string data type file the string data type right.

So, file is copied into file name that means file name equal to file right. So, it is having one display function right. So, this has to be defined because right. So, here you have image. that is a abstract class and I have real image I have the real image.

So, the display which is defined in line number 12. So, the display which is defined in line number 12. So, let us have the class proxy image right class proxy image which is inheriting image all right. So, this is also having at line number 20 you have display.

So, that means this proxy is inheriting image line number 20 you have the same display function right. So, real image the pointer object whose class is real image and you have file name string right. So, now you are the constructor proxy image file right. under string. So, file is copied into file name and real image is pointing to the null pointer right.

So, real image is pointing to null pointer real image is equal to null pointer right. So, void display what it is telling if not real image alright. So, not real image in general suppose right. So, let us say in the case of a main program we are going to see.

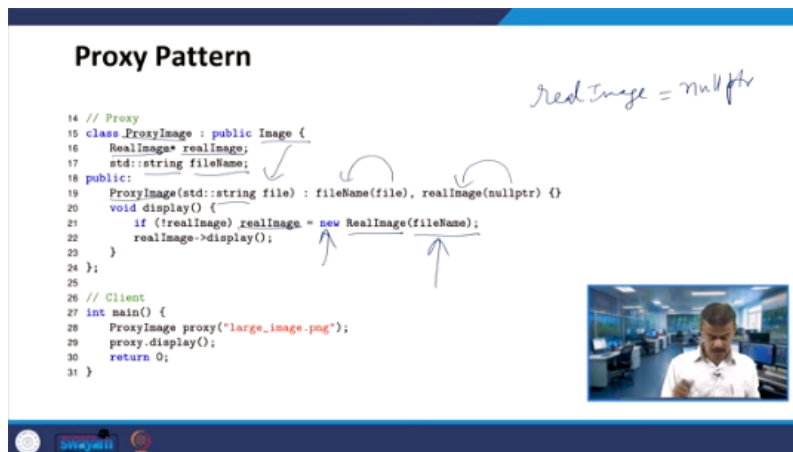
So, if real image which is not equal to null pointer. So, right real image is not equal to null pointer not real means it is pointing to right null. Then in that case you have right. So, you have right we are defining we are defining real image right memory allocating we are allocating a memory with the real image constructor. right, real image constructor by passing file name, right.

If I simply write if real image, that means real image not equal to null or not equal to null pointer, if I put not real image, right, so that means the real image is pointing to null pointer. So, we can take in that way, the opposite way, if I put not, all right. So, then the real image is, then the real image is invoking display, right. So, when the real image is invoking display, so here it goes, right. It will try to go to the superclass, that means, which is the abstract class, right?

Proxy Pattern

```
14 // Proxy
15 class ProxyImage : public Image {
16     RealImage* realImage;
17     std::string fileName;
18 public:
19     ProxyImage(std::string file) : fileName(file), realImage(nullptr) {}
20     void display() {
21         if (!realImage) realImage = new RealImage(fileName);
22         realImage->display();
23     }
24 };
25
26 // Client
27 int main() {
28     ProxyImage proxy("large_image.png");
29     proxy.display();
30     return 0;
31 }
```

Real Image = nullptr



And then line number 12 will be executed. Abstract class, it is abstract. In fact, it is a pure virtual function. So, therefore, line number 12 will be executed, right? So, let us see in the main program, in the client program, the main program.

So, we will see. I have an object called proxy, right? Under proxy image, right? The object called proxy by pausing large image dot png. I am pausing large image dot png, right?

So, then that means this constructor will be called. Now, file name is equal to file and real image is now pointing to the null pointer. Real image is now pointing to the null pointer. That is the meaning now, right? So, it will be understandable, right?

What is meant by if not real image. right so real image is pointing to null pointer right so now when proxy and in fact you are defining this object under proxy image so when i call this display right under proxy so right so here because the constructor right so this is being called so proxy is under proxy image So, proxy is under proxy image. So, this display suppose it is invoking yes in fact line number 29 it is invoking proxy is invoking display. So, this will be called if not real image right.

So, what is real image? Real image is null pointer right not real image is not null right the opposite one. not real image is not null pointer yes it is not null pointer because you have large image dot png it is pointing to large image dot png right so now this real image right is equal to now the real image of the file name right so the real image of the file name so file name is what large underscore image dot png so that means the constructor of real image will be called, you are passing large underscore image dot png, all right. So, here you go, real image constructor, file is copied, right.

What is file you are passing? You are passing large underscore image dot png, you are passing. So, that means, your file name is equal to, all right. So, file is nothing but, here file name, the file name is nothing but large image dot png, So, now your file name here in this case the file name is equal to large underscore image dot png ok.

So, still display function is there real image is now invoking display. So, when it is invoking display it goes to the superclass which is the abstract class right which is a pure virtual function. So, therefore, it is going to the subclass called real image. right so the real image subclass it is displaying right what is the file name large image dot png large underscore image dot png right so that will be

displayed so this is what we are expecting displaying will be printed followed by the name of the file name is large underscore image dot p Right.

So, here when I am talking about if not real image. So, real image is pointing to null pointer. So, not real image is the opposite. It is not null pointer. Okay.

So, that point you have to be clarified. So, suppose because here I do not know what exactly I am. So, suppose you are not pausing anything. Right. It is equal to null kind of.

So, nothing will happen. Right. So, I mean in fact it will not call any constructor. So, it is a null it will not call any constructor. So, here in this case I am passing this.

So, therefore, the real image initially pointing to null now is not null. This statement is not null, which is true. So, this is all about the proxy pattern, ok. So, this is a very good example of the proxy pattern. So, we have the bridge pattern, alright. So, the bridge pattern separates an abstraction from its implementation.

And also, it decouples abstraction and implementation for independent evolution. Right. So, an example is separating shape logic from rendering logic. And similarly, you have the flyweight pattern. So, you can see the example.

So, I have done several patterns. So, what you can do now. So, take these as an assignment and try to implement the bridge pattern and the flyweight pattern. Right. So, the purpose of the flyweight pattern is to minimize memory usage by sharing as much data as possible.

So, particularly this flyweight pattern is useful for large number of objects and reusing character objects in text centering is an example. So, these two what I am giving as an assignment you try all right because we have done several patterns. So, the bridge pattern and flyweight pattern. So, what you can do? So, you can try those as an assignment problem.

So, in the case of a structural patterns, so the real world applications are these adapter, decoder, composite, proxy and the bridge all right. So, in the case of general right so when i compare singleton factory observer and structural pattern so these are all we had seen right so the singleton pattern the last few classes back we had seen so this is ensuring a single instance globally the coffee

machine example that i had given so this is best for managing a shared resources all right and then we talked about factory pattern right in fact the coffee factory example we had seen So, this is providing an interface for creating objects right. So, best for decoupling object creation from usage and we started with observer pattern.

In fact, the last class we had talked about the observer pattern right. So, it is implementing a one to many relationship. We had seen the example in the case of both C++ and Java right. This is best for the dynamic update in even driven systems and we have seen many examples on structural patterns. So, this is organizing classes and objects to form flexible structures and this is for best for managing relationship between the entities.

For example, the adapter, composite, the decorator, right. So, these are all the examples we had seen, right. Adapter we had seen adapter pattern, composite pattern and decorator pattern, right. So, which pattern you will choose, which is the right pattern to choose for your application, right. I will use singleton, right, when we need a single instance, right, across the application.

So, logging, configuration, caching, these are all the examples. And when I want to have a database connectors, right, or any user interface components, so I choose the factory, right. So, this will be used when object creation varies and needs decoupling, right. And when I want to have the event driven systems or the real-time monitoring, right i use the observer pattern and when i have right for example i have the adapters composite and decorator right so when i want all this i use the structural pattern right so these are all the right way to choose singleton factory observer and structural patterns right so what are all the common misuse so you should not overuse the patterns right for the simple scenarios

I mean, one has to be very careful, right? The overuse of patterns or misapplying patterns in an appropriate problem, creating unnecessary complexity in the code, right? And the next point is, suppose, right, the ignoring scalability and performance considerations. So, this is another common misuse. Right.

So finally, failure to adapt patterns to specific applications. So these are all the common misuses of the design patterns. So what are all the best practices for implementing design patterns? Right. So we have to understand the problem.

So I'm going to choose which pattern. Right. So first understand the problem and choose the pattern and keep the implementation simple and focused. Right. So this is the best practice.

Another best practice. and use design patterns as guidelines not the strict rules right. And we can combine the patterns we have seen several patterns we can combine the patterns, but if necessary for complex problems right. So, before using this the design patterns you have to regularly review and refactor the code for optimal use of the patterns. So, these are all the best practices for implementing the design patterns.

So now, what are all the important aspects of design patterns in modern software development, right? So, I talked about code reusability, readability, and scalability, right? So, design patterns improve these, and standardized solutions enhance collaboration among developers, right? And these design patterns, Right.

Provide a blueprint for solving common design challenges. So, these patterns are essential for building robust, maintainable, and future-proof systems. All right. So, these four points are very important, and we call them the importance of design patterns in particular. So, they are useful for modern software development.

So, with this, I'm concluding this chapter. So, my advice is to try to implement all the design patterns. So, whatever design patterns I have given for practical problems, take them up, try to solve them, apply them, and get the output. So, with this, I am concluding this design pattern chapter. Thank you very much.