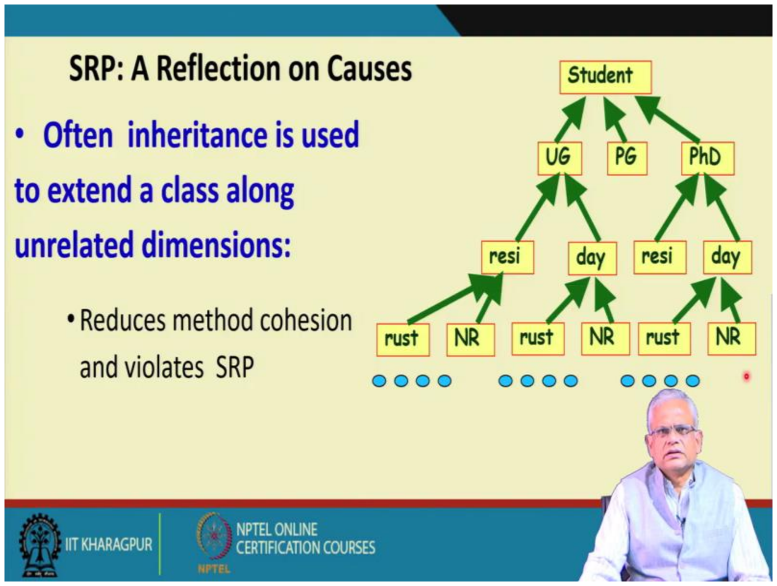
Object-Oriented System Development Using UML, Java and Patterns
Professor Rajib Mall
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
Lecture - 36
DIP Principle

Welcome to this session. In the last session, we were looking at a principle that is the single responsibility principle. The principle states that a class should have only one responsibility or a single responsibility. If a class supplies or is responsible for multiple things, then it is not cohesive and it will have different reasons to change, and the features that are implemented by the class gets coupled together and it becomes very difficult to maintain and the class becomes complex. And as a solution to the SRP problem, we said that if a class is not compliant to SRP we should use delegation.

But the thing is that when we start writing the code and if you are not really conscious of this principle, a class gradually, over the maintenance gets multiple responsibilities. Let us just take an example.

(Refer Slide Time: 01:42)



SRP: A Reflection on Causes

- Often inheritance is used to extend a class along unrelated dimensions:
  - Reduces method cohesion and violates SRP

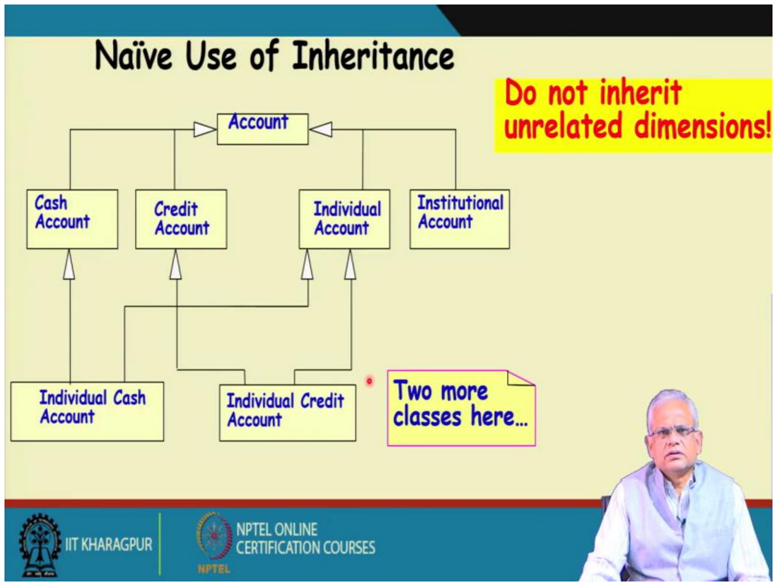
$$\begin{tikzcd}
 & & & \text{Student} & & & \\
 & \text{UG} \arrow[urr] & & \text{PG} \arrow[u] & & \text{PhD} \arrow[ull] & \\
\text{resi} \arrow[ur] & & \text{day} \arrow[ul] & & \text{resi} \arrow[ur] & & \text{day} \arrow[ul] \\
\text{rust} \arrow[u] & \text{NR} \arrow[ul] & \text{rust} \arrow[u] & \text{NR} \arrow[ul] & \text{rust} \arrow[urr] & & \text{NR} \arrow[u]
\end{tikzcd}$$

The main problem here is that once we start writing the code, often inheritance is used to extend a class along unrelated dimensions and this reduces the cohesion of the class and results in a violation of the circle. Let us look at one simple example, we had a Student class and then we had different types of students UG, PG, PhD, this also worked fine.

But then we had this complication that we had some of the students residing in the hostels and some are day scholars. And for PG also, similarly, we have resident and day scholar. And after some time, it became necessary to implement another fact that some students are rusticated for a semester or a year and we need to treat these objects differently, the rusticated students, because they have the hostel but they have been rusticated or they are day scholar and rusticated and the students which are non-rusticated, and just see here there is explosion of the class hierarchy and this is a symptom of extending the class in different dimensions.

How do we handle this problem? The answer is that we need to delegate. But how do we apply it to this problem to delegate? We will just see that briefly. There can be other reasons for change, and so on, and the hierarchy becomes still worse.

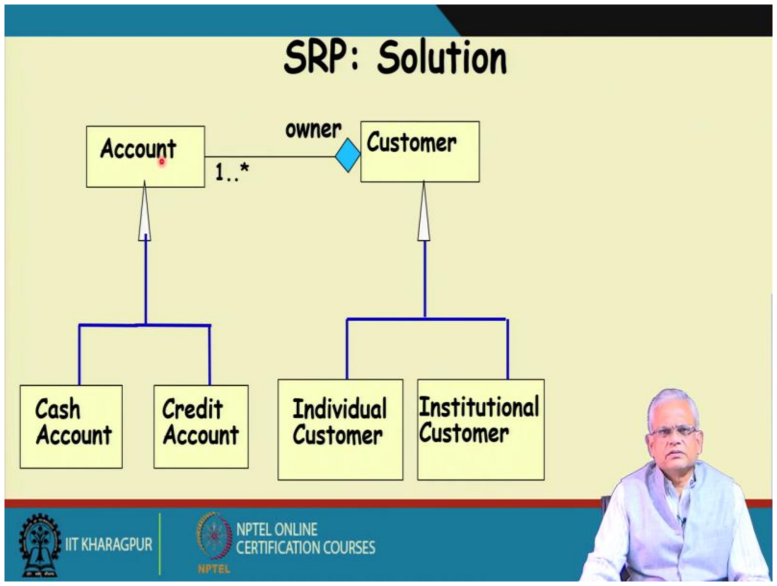
(Refer Slide Time: 03:52)



That is another example here that we had an Account class in a bank and then we had two types of account, the cash accountant and credit account. But then, the account can belong to an individual or it can be an institution or a company. There are two types of account, there is some difference between these two types of accounts because, in an institutional account, there is some signatories who may change and so on.

But then we heard the cash account and credit account and we need to have individual cash account and individual credit account; the institutional credit account, institutional cash account, and so on. But then what is the solution?

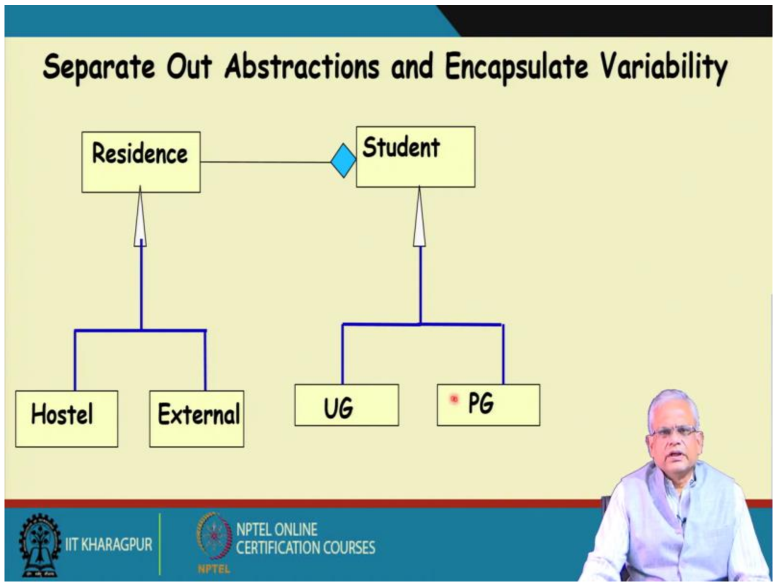
(Refer Slide Time: 05:05)



SRP: Solution

The solution is that we have this Account class. Basic account class, there are two types of account, the cash account, and credit account, and the customer can own many accounts. And the customer can be an individual customer or an institutional customer, and therefore, we separate out these different hierarchies. And now, each of the class has a single responsibility and that we achieve by delegation, the customer delegated the account to a different Account class in the previous solution that we just discussed, we had all this merged entangled.

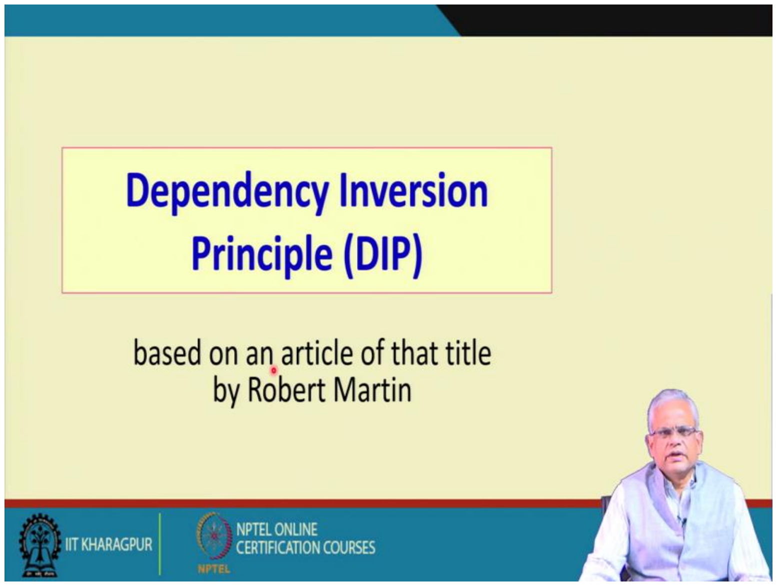
(Refer Slide Time: 06:03)



Separate Out Abstractions and Encapsulate Variability

Similarly, the student problem we have the different types of students UG, PG, research, and so on and we need to delegate the residence to a hostel or external.

(Refer Slide Time: 06:24)

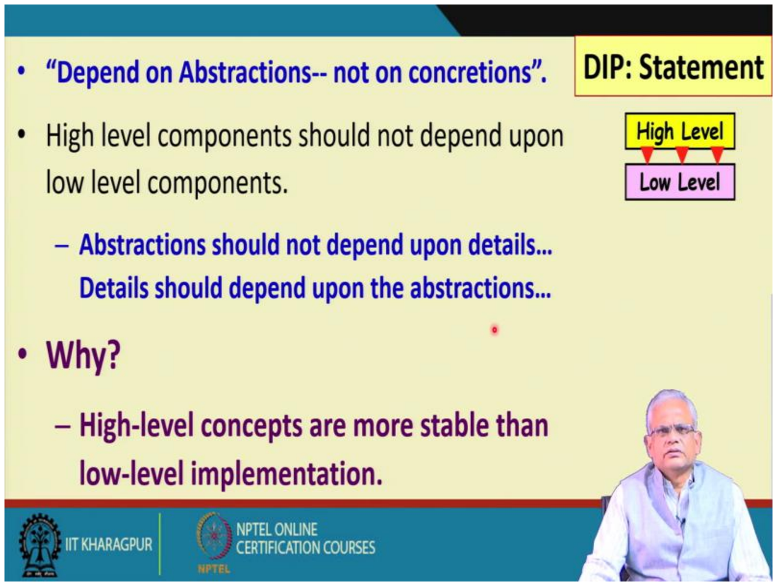


Dependency Inversion Principle (DIP)

based on an article of that title by Robert Martin

If we do not solve the single responsibility principle, if there is a violation of the single responsibility principle, we must solve it by delegation, otherwise, the classes become unnecessarily complex and there will be too many changes, and the clients will need to change even though, if we had implemented the SRP, the classes would not have to change. Now, let us look at the last principle of the SOLID principle, the five principles we were mentioning. This is the last one, the dependency inversion principle or DIP. This is again based on an article by Robert Martin.

(Refer Slide Time: 07:13)



DIP: Statement

- "Depend on Abstractions-- not on concretions".
- High level components should not depend upon low level components.
  - Abstractions should not depend upon details... Details should depend upon the abstractions...
- Why?
  - High-level concepts are more stable than low-level implementation.

High Level
Low Level
IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES

Before we can discuss the DIP principle, different dependency inversion principle, let us see that a typical software is structured into layers. There are high-level features implemented in some high-level classes and then there are some low-level features and typically, the high-level features invoke features of the low-level classes

But then, the high-level features should not really depend on the low-level features. If we had an implementation like the high-level features directly calling the low-level classes, the high-level classes calling the low-level classes, then we will have a violation of the dependency inversion principle, because that will lead to many changes in the code and complexity.

The statement of the problem says that the high-level components should not depend upon the low-level components. But just see here that the high-level components or the high-level classes are calling the methods of the low-level classes that take the help of the low-level classes and they become dependent. Typically, that happens if we are not careful, we are not aware of this principle, and the solution by which we make the high-level classes independent of the low-level classes is by using the abstract classes and interfaces. And the solution here is to depend on abstractions and not on concretions.

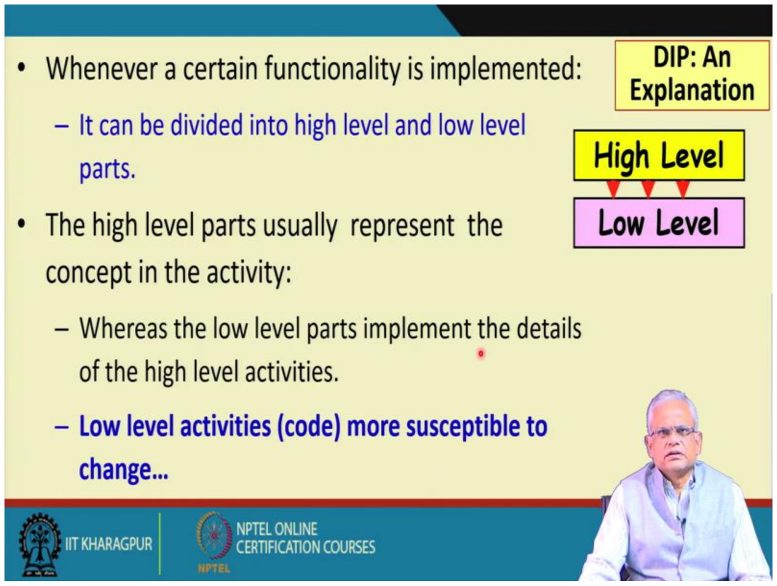
If the high-level classes called the methods of the low-level classes directly i.e., the concrete classes, then we will have a violation of the DIP. This would actually call methods and the interfaces are abstract classes. Another statement of the principle is that abstractions should not depend on details; the details should depend on the abstractions.

We can think of the high-level components as abstractions of the low-level components because this deal with some higher-level issues and the low-level ones, they do deal with some specific implementations in a typical layered solution and it is wrong if the abstractions depend on the concrete or the details. It should be that the details should dependent on the abstractions and not vice versa.

Now, why is the violation of DIP not a good idea? What problems might occur? The problem is that the low-level classes or components, they change frequently. For example, the GUI, they change frequently. Now if a change of GUI needs to change the entity classes that is not a good idea, the entity classes are more stable.

The high-level concepts are more stable, they do not change, but the lower-level implementations, they do change. For example, the IO classes or the GUI classes they are at the lowest level and they do change depending on the platform, and so on. And this should not force a change of the high-level classes but then we need to understand that how to implement or how to make a design compliant with DIP.

(Refer Slide Time: 11:41)



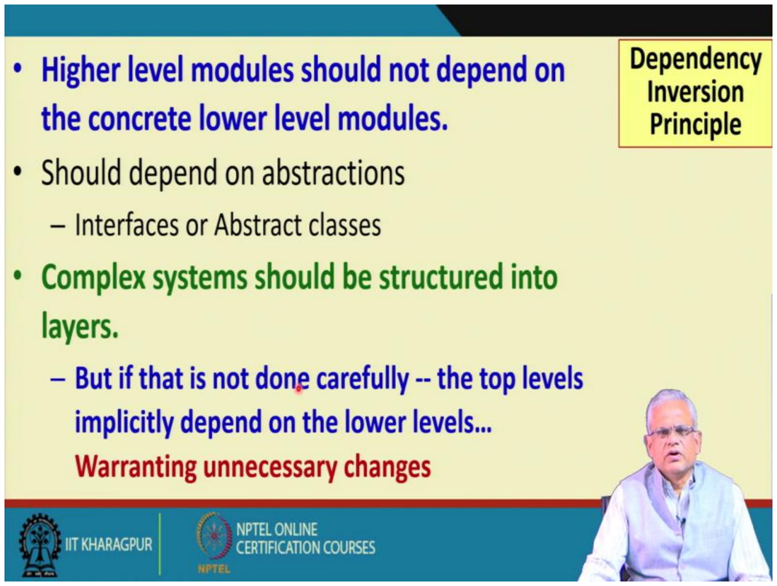
DIP: An Explanation

- Whenever a certain functionality is implemented:
  - It can be divided into high level and low level parts.
- The high level parts usually represent the concept in the activity:
  - Whereas the low level parts implement the details of the high level activities.
  - **Low level activities (code) more susceptible to change…**

High Level
Low Level
IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES

Most of the solutions that we have if you analyse that you will find that such a situation exists that the high-level classes, this is the topmost layer. And then we have the low-level classes which provide some specific services to the high-level classes. But then, unless we are careful, the high-level classes become dependent on the lower level classes. If they directly call the concrete classes, they become dependent on it and the solution quality degrades and it becomes a poor-quality program. The high-level parts represent the concepts, the low-level implement the details, and the low-level activities are more susceptible to change.

(Refer Slide Time: 12:50)



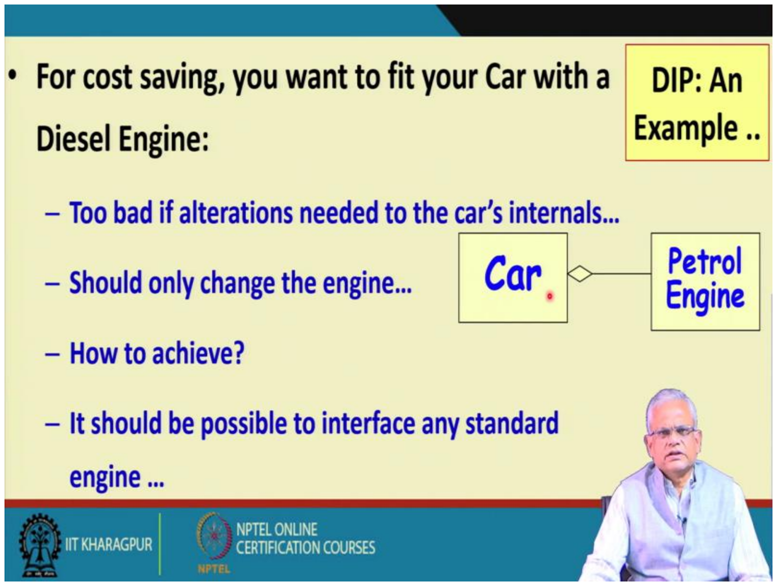
Dependency Inversion Principle

- **Higher level modules should not depend on the concrete lower level modules.**
- Should depend on abstractions
  - Interfaces or Abstract classes
- **Complex systems should be structured into layers.**
  - **But if that is not done carefully -- the top levels implicitly depend on the lower levels... Warranting unnecessary changes**

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

But what is the solution? We need to use again interfaces or abstract classes, and whenever we layer all designs, good designs are layered designs; that is a good thing if a solution is layered, but we have to be extra careful. We should make sure that the top-level or the high-level classes do not directly depend on the low-level classes, otherwise, the solution becomes inferior and there will be too many changes bugs, and so on.

(Refer Slide Time: 13:37)



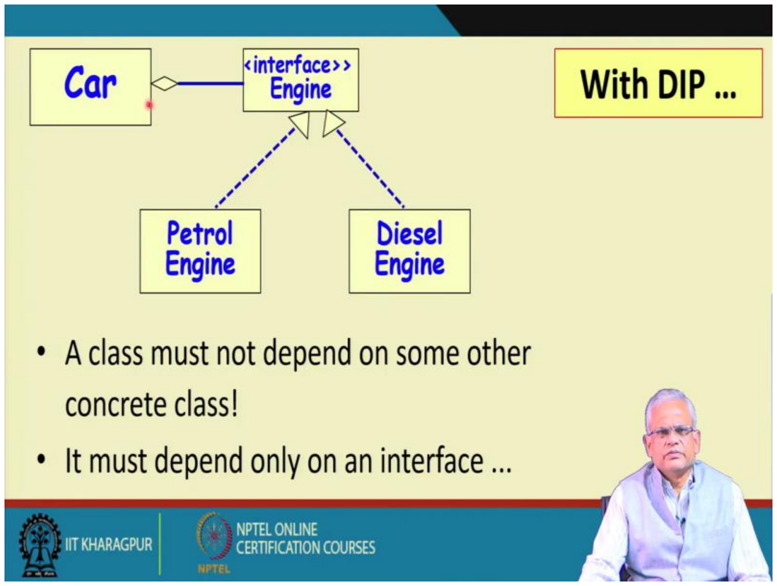
DIP: An Example ..

- For cost saving, you want to fit your Car with a Diesel Engine:
  - Too bad if alterations needed to the car's internals...
  - Should only change the engine...
  - How to achieve?
  - It should be possible to interface any standard engine ...

Let us just look at an example and we will also look at the solution. Hypothetical example, just to illustrate the principle. Let us consider a car has a petrol engine and other components. Now, let us say that petrol is expensive and we decided to change the petrol engine with a diesel engine. Now, in the changing process, if we want to take out the petrol engine and make several alterations inside the car, then our design is not good. We should be able to just take out the petrol engine and seamlessly put the diesel engine and then it is a good design. We should only change the engine without really changing the car. But how do you achieve it?

The key here is if you observe that the different engines should have a standard interface and the different engines petrol engine, diesel engine, gas engine, etc, they should all implement that interface and the car should invoke the service of the engine through this interface.

(Refer Slide Time: 15:27)



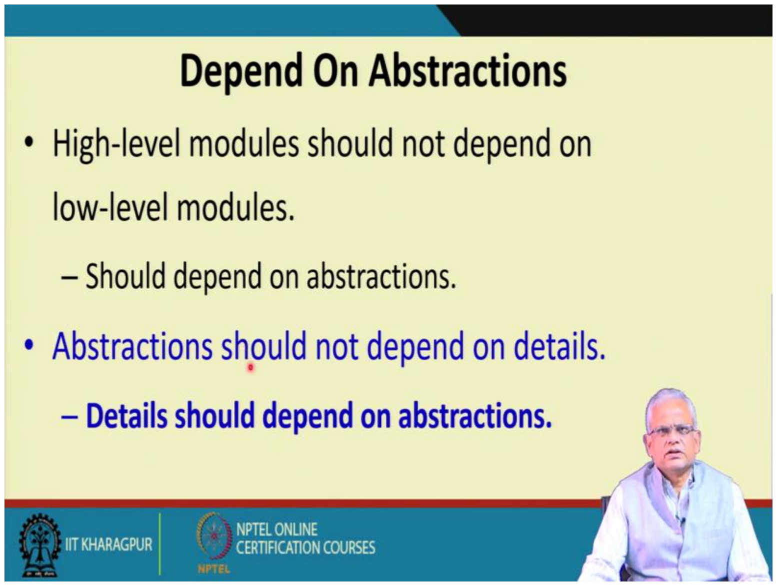
With DIP ...

- A class must not depend on some other concrete class!
- It must depend only on an interface ...

If we redraw this, it should be like this, that a car can have either a petrol engine or diesel engine which implement the engine interface? In this situation, we can take out the engine and seamlessly have the other type of engine replaced. And this is the DIP compliant, the car is like a high-level concept, the engine is like a low-level and the car should not depend on the engine if the engine, they implement the engine interfaces, the petrol engine and diesel engine implement the engine interface.

The problem was arising because the car was fitted directly with engine, a petrol engine and the petrol engine was not really implementing an interface and in that case, it becomes very difficult to take out the petrol engine and then have to make modifications to the car internals to have the diesel engine fitted. So the solution to the dependency inversion principle is to have an interface. The low-level components must implement an interface and the high-level components just invoke the services through the interface.

(Refer Slide Time: 17:20)



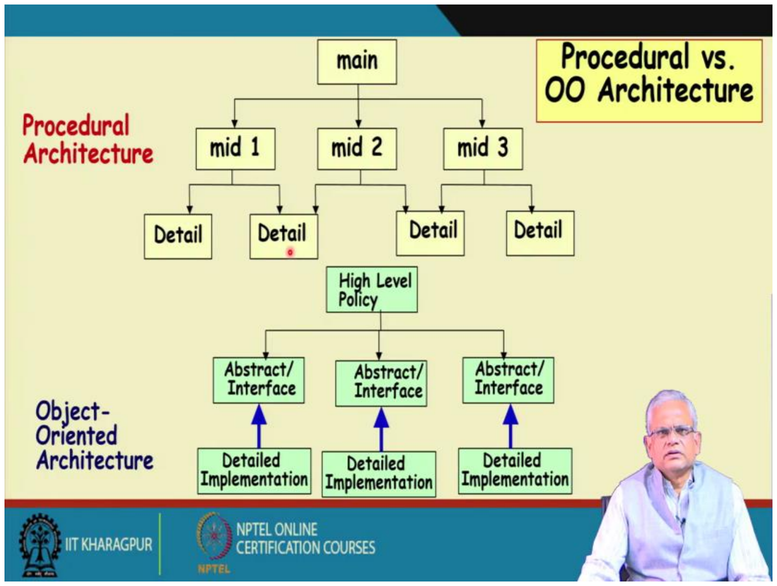
Depend On Abstractions

- High-level modules should not depend on low-level modules.
  - Should depend on abstractions.
- Abstractions should not depend on details.
  - **Details should depend on abstractions.**

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES

The high-level classes should not directly call the services of the lower-level classes, but they should call through abstractions, abstract classes, or interfaces. And this helps us to have the DIP compliant solutions where abstractions do not depend on the details and the details depend on abstractions.

(Refer Slide Time: 17:50)

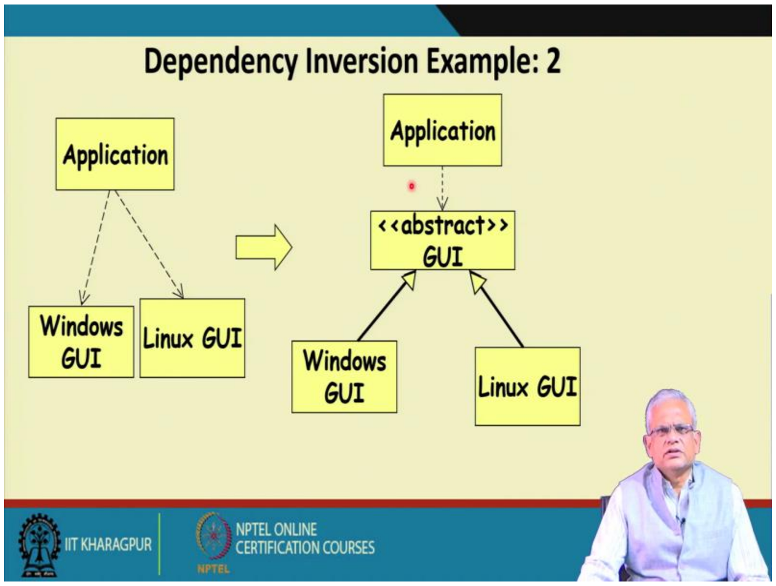


If we just reflect back on procedural design, this was very common. In a procedural design, we have the main program or the main module, which calls some mid-level modules and the mid-level modules calls the detailed modules which do I/O, and so on; the input/output aspects are handled by the detail. This was a very prevalent design in the procedural design but see here that all these, this design they violate the dependency inversion principle.

But in an object-oriented architecture, we have abstract classes here and the detailed implementations are inherited or implemented by the interface here. Inherit the abstract class or they implement the interface here, and the high-level classes are invoked through the interfaces or abstract classes. And therefore, the object-oriented architectures are good compared to procedural architectures. They isolate the changes in the detailed implementation to percolate to the high-level policy classes. These are highly stable classes and we should not unnecessarily change them when the detailed implementation classes change.

If we are careful with object-oriented architecture, we can achieve dependency inversion, which will be DIP compliant. But in a procedural design, it is not achievable. If you reflect why we cannot achieve DIP compliant code in procedural architecture, you will see that the main culprit here is that we do not have some concept like abstract classes or interface classes in procedural code and we cannot really achieve the dependency inversion principle.

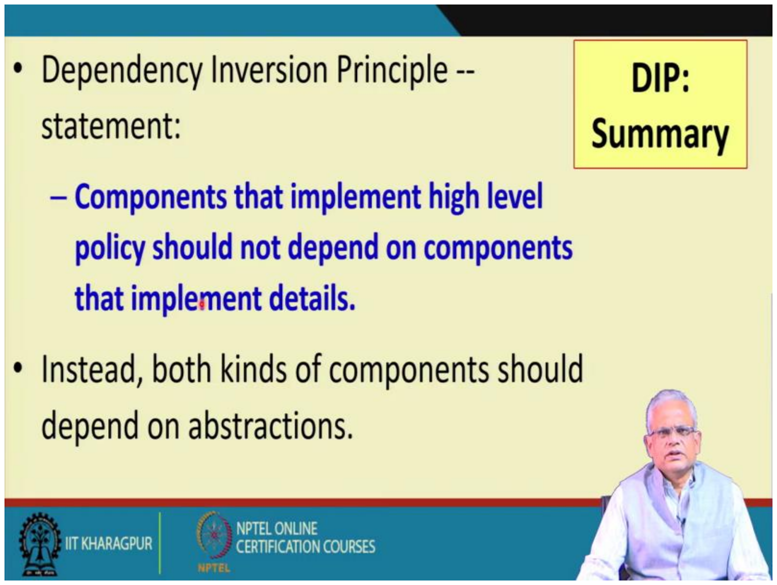
(Refer Slide Time: 20:28)



Let us look at another example. Let us say an application is calling methods of GUI. Let us say it works with Windows GUI and Linux GUI. Now, it needs to call the methods directly to the Windows GUI and the Linux GUI, then it is not really compliant to DIP because it was initially using the Windows GUI and then later we had to support the Linux GUI also, then we need to change the application and the Mac GUI and so on becomes difficult because each time we need to change the application. But how do we avoid that?

We will avoid that if the Windows GUI and the Linux GUI implement or inherit an abstract GUI class and then the application invokes the services through the abstract GUI, then unnecessary changes when we change the user interface style can be avoided.

(Refer Slide Time: 21:50)



DIP: Summary

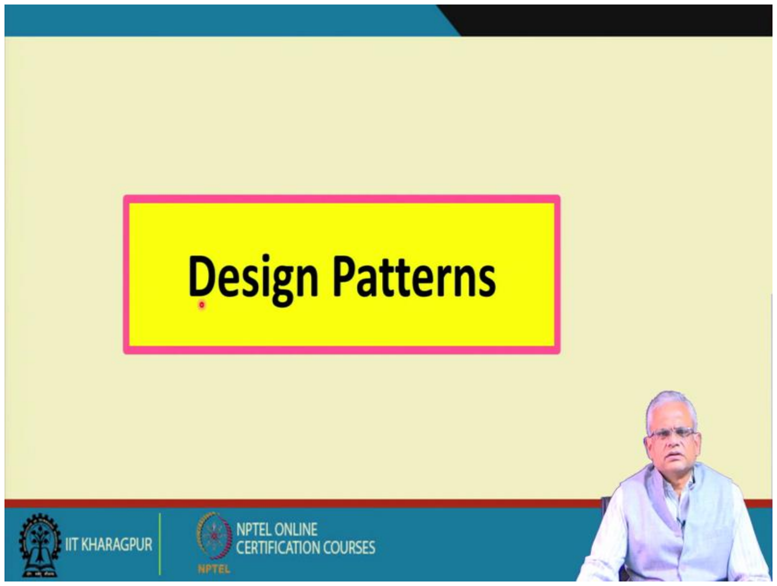
- Dependency Inversion Principle -- statement:
  - Components that implement high level policy should not depend on components that implement details.
- Instead, both kinds of components should depend on abstractions.

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES
NPTEL

Now, let us look at the summary of the dependency inversion principle. The components that implement high-level policy should not depend on components that implement the details. We saw the solution on how to achieve it. The service of the low-level components should be invoked through an interface class or abstract class and the low-level components must implement or be derived, implement an interface are derived from an abstract class.

We looked at all the five principles behind good designs and if you reflect back on these principles that we discussed over the last one session, you will see that most of the solution is in the form of using abstract classes and interface classes. And we will see that the design pattern solutions, they use the interface classes and abstract classes very profusely and that is justified based on these five principles. As we look at the design pattern solutions, will recollect that what are the design principles and the pattern solutions are compliant to these principles.

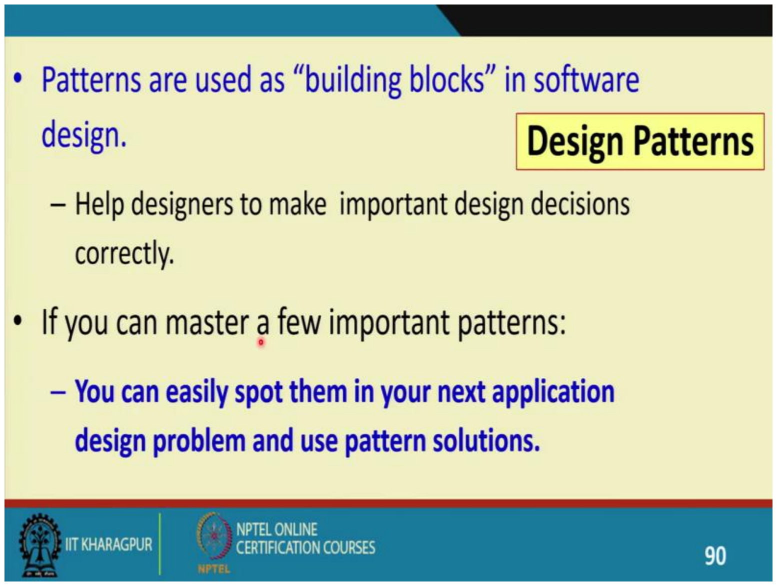
(Refer Slide Time: 23:46)



Design Patterns

Now, let us discuss the design patterns.

(Refer Slide Time: 23:57)



Design Patterns

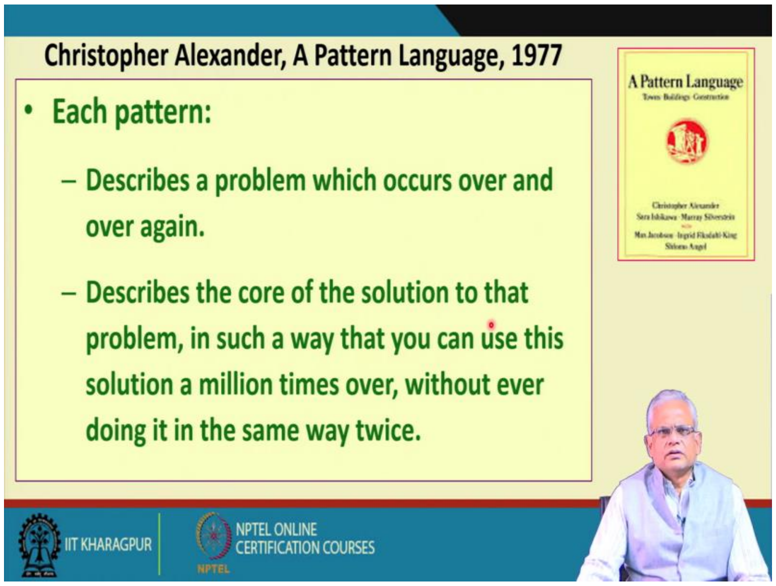
- Patterns are used as "building blocks" in software design.
  - Help designers to make important design decisions correctly.
- If you can master a few important patterns:
  - **You can easily spot them in your next application design problem and use pattern solutions.**

90

What is a design pattern? You can think of a design pattern as the building blocks of software design. If we know the pattern solutions which are good solutions somebody has developed, we have good programmers, who come up with very good solutions and if we can somehow learn those solutions, then we can reuse them.

An application is not reusable in its entirety, but small parts of the application become reusable, that we call as the building blocks. If we know these reusable building blocks, we can develop another application by making use of these different building blocks and we can effortlessly arrive at a good design. That is the main motivation here because it helps us reusing the good designs. If we can master some important patterns when you are trying to solve an application problem, we can easily spot them and then reuse the design pattern solution and that will lead us to a high-quality design.

(Refer Slide Time: 25:29)



Christopher Alexander, A Pattern Language, 1977

- Each pattern:
  - Describes a problem which occurs over and over again.
  - Describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it in the same way twice.

A Pattern Language
Towns Buildings Construction
Christopher Alexander
Sara Ishikawa · Murray Silverstein
Max Jacobson · Ingrid Fiksdahl-King
Shlomo Angel
IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES
NPTEL

The origin of the design pattern is from Christopher Alexander. He wrote this book, A Pattern Language. He was not a computer scientist; he was an architect. He used to design buildings but then he heard this idea that different designs of buildings, there are some things which can be codified into patterns and if somebody can master this patterns, then he can effortlessly design a new building by falling back on those patterns. He encoded some 300 odd patterns in his Pattern Language book.

Each of those 300 odd patterns describes a problem which occurs over and over again in different applications to be designed or different buildings to be designed for each case. It describes the core of the solutions to that problem in such a way that you can use the solution a million times over without ever doing the same way twice.

This is very important statement here that the solutions that are provided in the patterns when we reuse it in a different situation to solve a different application, we do not take it and plug it, we use it by adapting it. And therefore, it says that we can use the solution, the main ideas of the solution a million times over without doing the same way twice.

As we proceed with our design pattern discussions will see how to make use of a pattern solutions and we will see that each time we make some changes to the pattern. It is not like making a library call, a library routine call is a reuse where we just call each time the same way, but the design pattern solution is not like making a call to a library routine. Here, we take the central solution and adapt it to our purpose.

The work of Christopher Alexander, even though he was not a computer scientist, but his work has inspired the design pattern solution in object-oriented design. We will just see the details of the object-oriented design patterns in our subsequent sessions. We are at the end of this session, we will stop here and continue in the next session. Thank you.