S: Single Responsibility Principle: Each class should have only one purpose. It should not try to multitask any more than it has to. It's not to say each class can only have one function.

It's like saying that in an office, let the secretary be the secretary, the security guard be the security guard and the doorman be the doorman. Bad things happen if you try to combine all those duties and drop it on just one person.

O: Open Closed Principle: When you write a class or function or library you should do it in a way that anyone else can easily build on to it, but not change it's core elements.

If you are holding a barbecue and bring the steaks, always allow other people to bring things like condiments, salads, drinks, chips, knifes, forks, etc but never allow anyone to get rid of your steaks and replace them with veggie burgers (emphasis on replace).

L: Liskov's Substitution Principle: Simply put, any time you have a sub-type of something, that subtype should be 100% compatible with the original thing. This is usually not an issue since a subtype is a specialized version of the more generic thing.

If you have a rectangle class that has "width and "height" as properties and want to make a square class, don't use "side length" as it's property. You should be able to put in a rectangle where you would normally put a square and everything should work. Instead just make "width" = "height".

Alternatively: "If it looks like a duck, quacks like a duck, but needs batteries - you probably have the wrong abstraction".

I: Interface Segregation Principle: This one says that only make the user impliment whichever methods they intend to use. A problem commonly found in interfaces, which are a bundle of functions, which have a name, and defined inputs/outputs but no code, and need the user to make code for them, or else it won't compile. A bad interface is one that has too many functions for what the user wants. If you only intend to use a handful of relevent functions but there are 100 others, that's 100 do nothing code snippets that need to be added to your code. Instead you should split this up into it's most relevent groups.

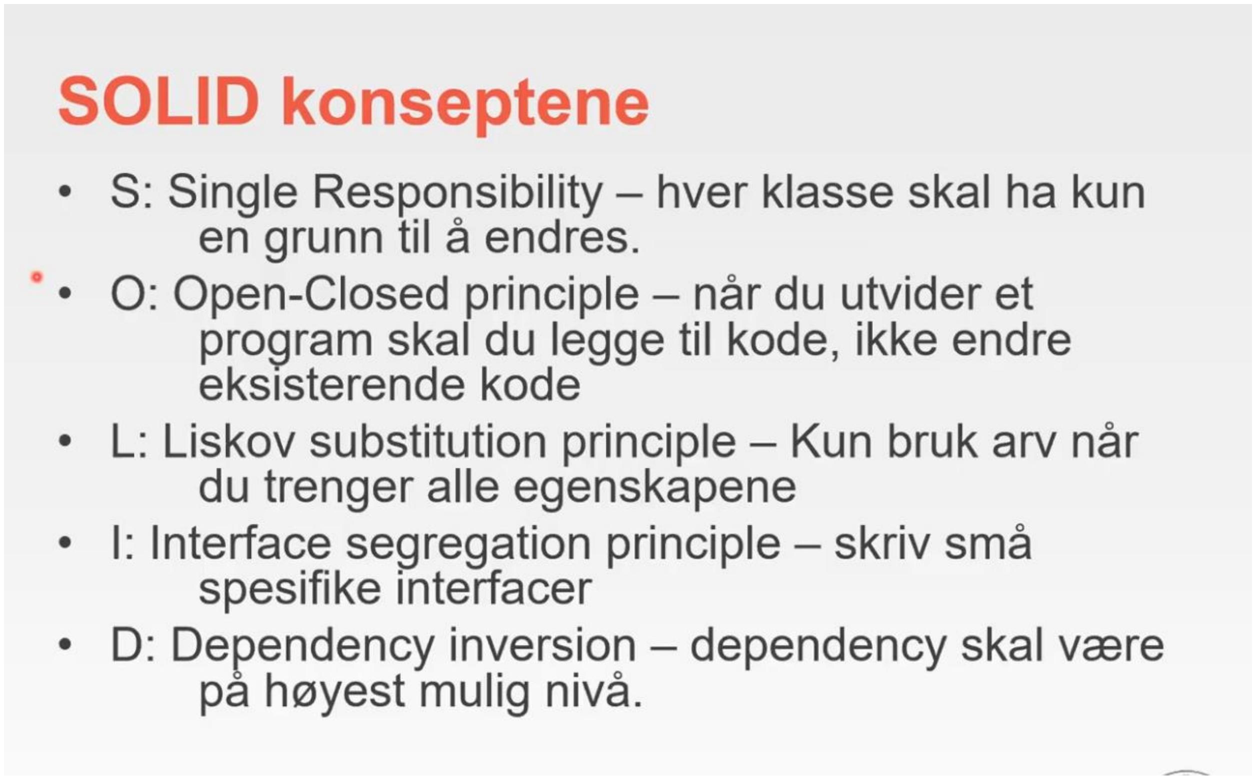
If you are a mechanic that only offered two services, a complete overhaul of the whole car top top bottom, or nothing at all, nobody would go to your shop. Instead you should offer a brakes package, exhaust package, oil changes, etc.

D: Dependency Inversion Principle: High level modules shouldn't depend on low level modules, both should depend on abstractions. Also abstractions should not depend on the small details, the details should depend on the abstractions. A good top level module should be flexible regarding how it's lower level modules operate and shouldn't have to assume a specific way of operating. Likewise a low level module should not be concerned with which specific top level module it's reporting to. Also both of these should not have details that are specific to one or the other, just a specification that will suit any application.

This is usually accomplished by using an interface between the high level and low level. The high level can call the same function on all of the low level modules, knowing that the input and output will always be of the expected type and the low level reports to the interface using the same logic.

Suppose you work at a company and you are a manager. You have a lot of things that need doing and a lot of employees. Those employees don't work only for you, they have other managers. Instead of teaching them how to do their reports specifically for your case, you should teach them the basic principles of that report, and do the formalization yourself. From an employee's point of view, It gets overwhelming to memorize every managers preferences. It would be much easier if they can just write a generic report for everyone with a certain set of data. In this case the generic report process is the interface. All you need is the data and you know what the output should look like.

By doing this, you as a manager aren't reliant on having your employees give their reports in a super specific fashion (you'd have to proof read anyway). You as a manager can simply give them the data and get cracking. If those employees have to give that report to anyone else, they won't be confused by the twists you put on yours. As an employee it allows you to work for multiple managers using only a simple set of rules, rather than memorizing a huge stack of little details. Everyone wins.



SOLID konseptene

- S: Single Responsibility – hver klasse skal ha kun en grunn til å endres.
- O: Open-Closed principle – når du utvider et program skal du legge til kode, ikke endre eksisterende kode
- L: Liskov substitution principle – Kun bruk arv når du trenger alle egenskapene
- I: Interface segregation principle – skriv små spesifike interfacer
- D: Dependency inversion – dependency skal være på høyest mulig nivå.