Principles of software engineering

Separation of Concerns

Separation of concerns is a recognition of the need for human beings to work within a limited context. As descibed by G. A. Miller [Miller56], the human mind is limited to dealing with approximately seven units of data at a time. A unit is something that a person has learned to deal with as a whole - a single abstraction or concept. Although human capacity for forming abstractions appears to be unlimited, it takes time and repetitive use for an abstraction to become a useful tool; that is, to serve as a unit.

Modularity

The principle of modularity is a specialization of the principle of separation of concerns. Following the principle of modularity implies separating software into components according to functionality and responsibility.

Abstraction

The principle of abstraction is another specialization of the principle of separation of concerns. Following the principle of abstraction implies separating the behavior of software components from their implementation. It requires learning to look at software and software components from two points of view: what it does, and how it does it.

Failure to separate behavior from implementation is a common cause of unnecessary coupling.

Principles of software engineering

Anticipation of Change

Computer software is an automated solution to a problem. The problem arises in some context, or domain that is familiar to the users of the software. The domain defines the types of data that the users need to work with and relationships between the types of data.

Working out an automated solution to a problem is thus a learning experience for both software developers and their clients. Software developers are learning the domain that the clients work in. They are also learning the values of the client: what form of data presentation is most useful to the client, what kinds of data are crucial and require special protective measures.

If the problem to be solved is complex then it is not reasonable to assume that the best solution will be worked out in a short period of time. The clients do, however, want a timely solution. In most cases, they are not willing to wait until the perfect solution is worked out. They want a reasonable solution soon; perfection can come later. To develop a timely solution, software developers need to know the requirements: how the software should behave. The principle of acticipation of change recognizes the complexity of the learning process for both software developers and their clients. Preliminary requirements need to be worked out early, but it should be possible to make changes in the requirements as learning progresses.

Consistency

The principle of consistency is a recognition of the fact that it is easier to do things in a familiar context. For example, coding style is a consistent manner of laying out code text. This serves two purposes. First, it makes reading the code easier. Second, it allows programmers to automate part of the skills required in code entry, freeing the programmer's mind to deal with more important issues.

Principles of software engineering

Generality

The principle of generality is closely related to the principle of anticipation of change. It is important in designing software that is free from unnatural restrictions and limitations. One excellent example of an unnatural restriction or limitation is the use of two digit year numbers, which has led to the ”year 2000” problem: software that will garble record keeping at the turn of the century. Although the two-digit limitation appeared reasonable at the time, good software frequently survives beyond its expected lifetime.

Incremental Development

Fowler and Scott [FS97] give a brief, but thoughtful, description of an incremental software development process. In this process, you build the software in small increments; for example, adding one use case at a time.

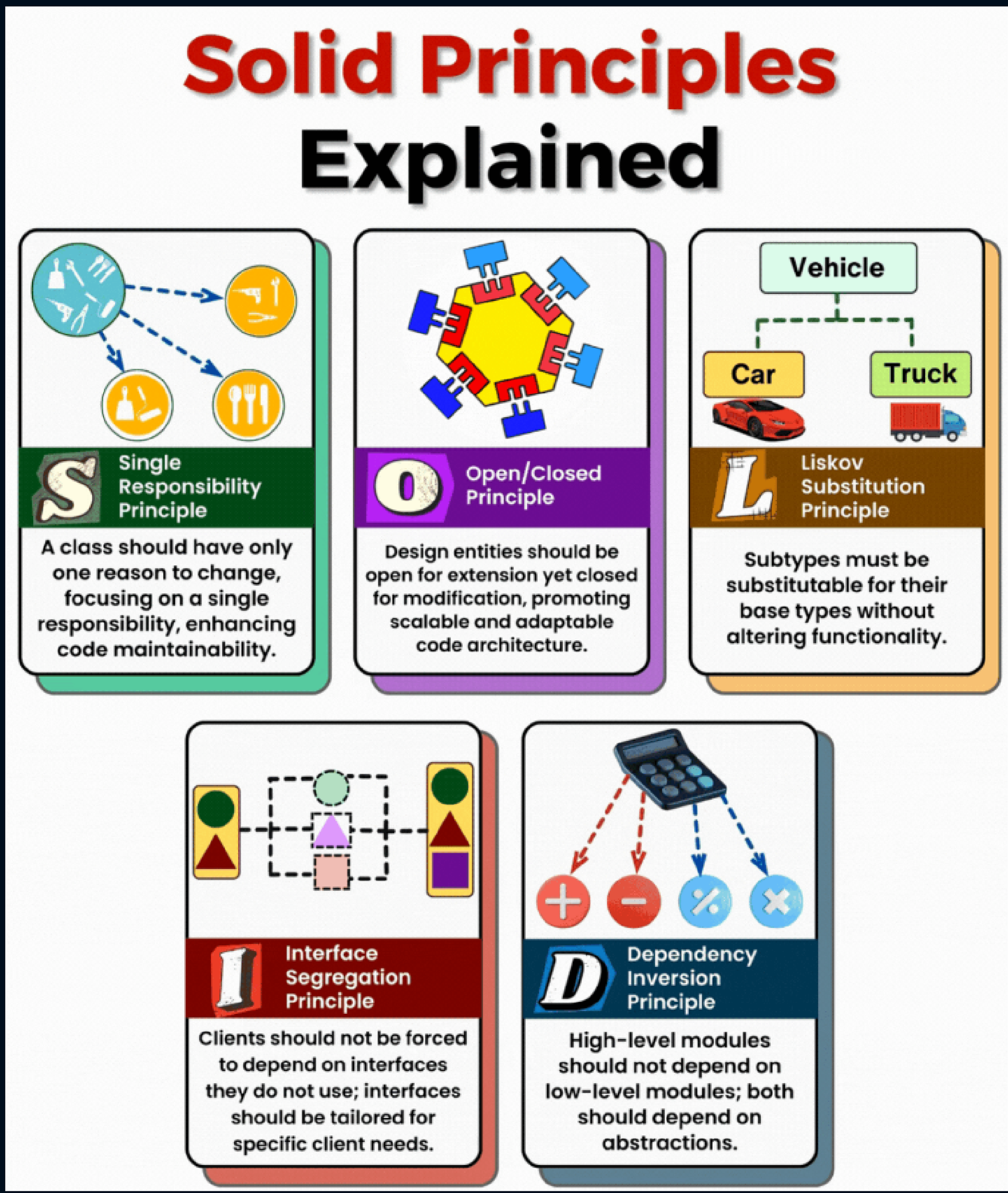
An incremental software development process simplifies verification. If you develop software by adding small increments of functionality then, for verification, you only need to deal with the added portion. If there are any errors detected then they are already partly isolated so they are much easier to correct.

Incremental Development

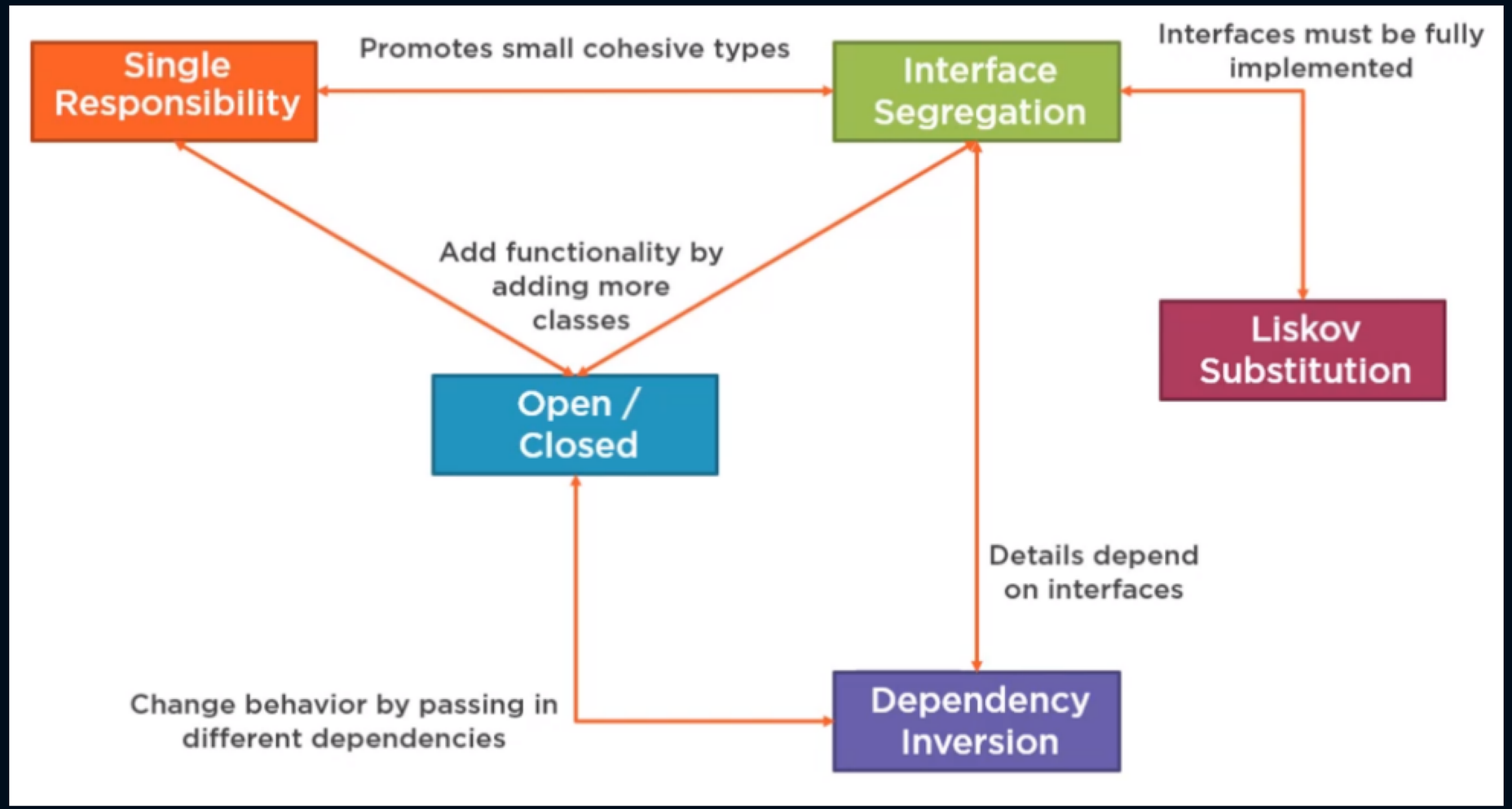
Fowler and Scott [FS97] give a brief, but thoughtful, description of an incremental software development process. In this process, you build the software in small increments; for example, adding one use case at a time.

An incremental software development process simplifies verification. If you develop software by adding small increments of functionality then, for verification, you only need to deal with the added portion. If there are any

SOLID Principles



SOLID Principles Interconnection



SOLID TLDR

SRP: Every class or module in a program should have **Responsibility** for just a **Single** piece of that program's functionality

OCP: Software entities should be **Open** for extension, but **Closed** for modification.

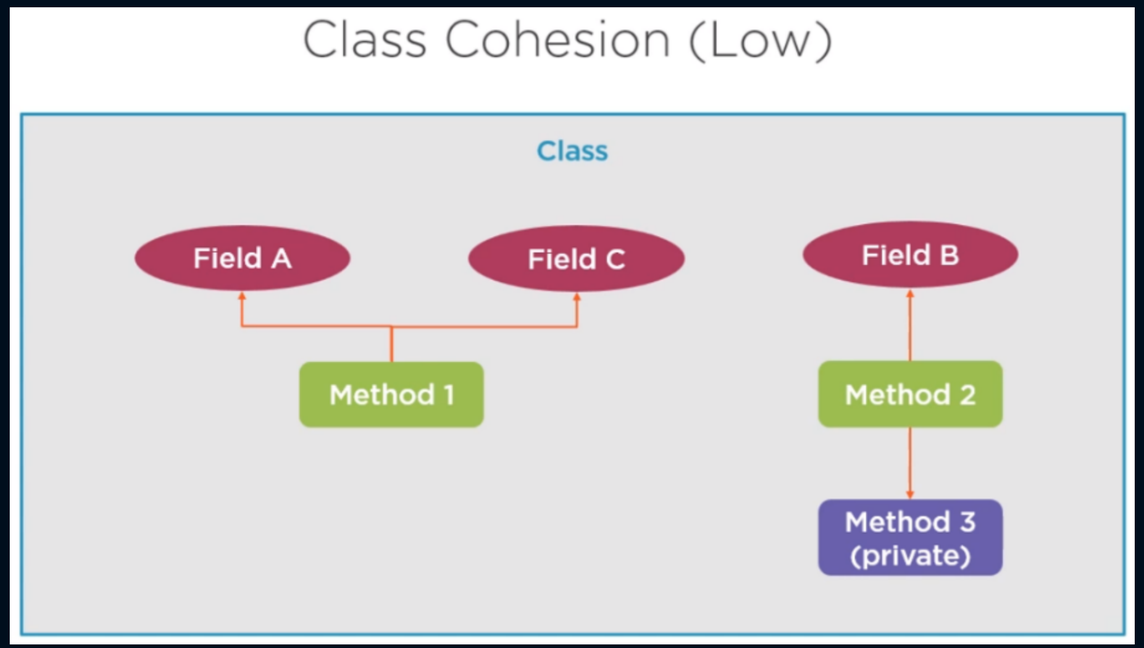
LSP: Objects should be replaceable with instances of their subtypes without altering the correctness of that program,

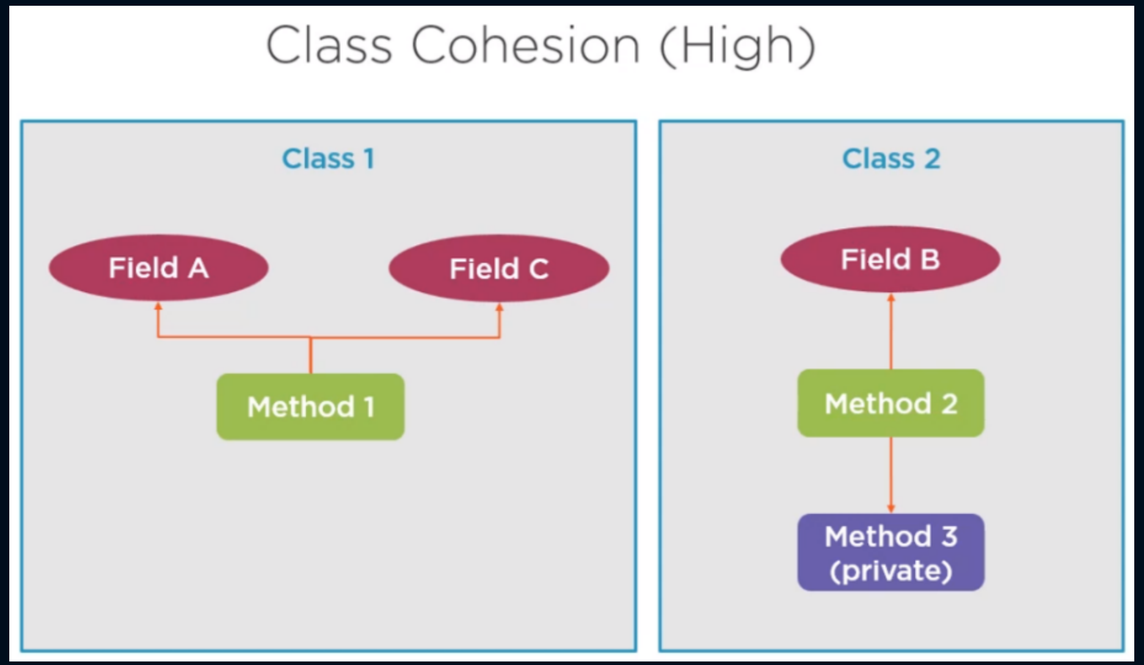
ISP: Splits interfaces that are very large into smaller and more specific ones so that implementations will only have to know about the methods that are of interest to them.

DIP: High level modules should not depend on low level modules. Both should depend on abstractions

SRP: Cohesion

In the context of asynchronous programming, it's essential to avoid using async void methods unless they are event handlers or top-level entry points (e.g., Main method in console applications or event handlers in GUI applications). The reason is that async void methods are not easily awaited, and exceptions thrown within them cannot be caught directly by the calling code.

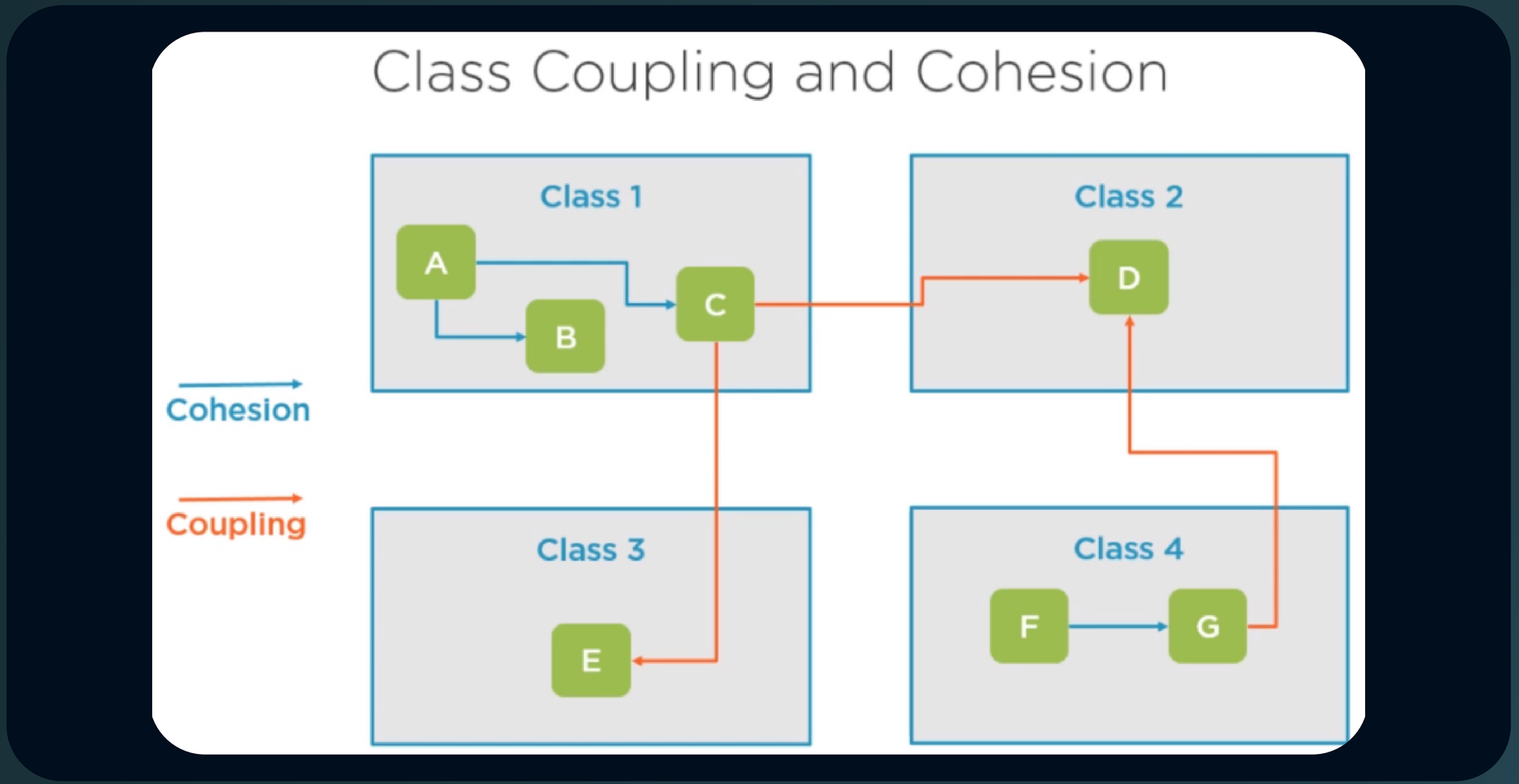




SRP: Coupling

Tight Coupling: Binds 2 or more details together in a way that is difficult to change.

Loose Coupling: Offers a modular way to choose which details are involved in a particular operation.



OCP: Abstraction and concreteness

Start concrete, modify the code the first time or two and in the third time consider making the code open for extension. Predict where variation is needed and apply abstraction as needed

Extreme Extensibility

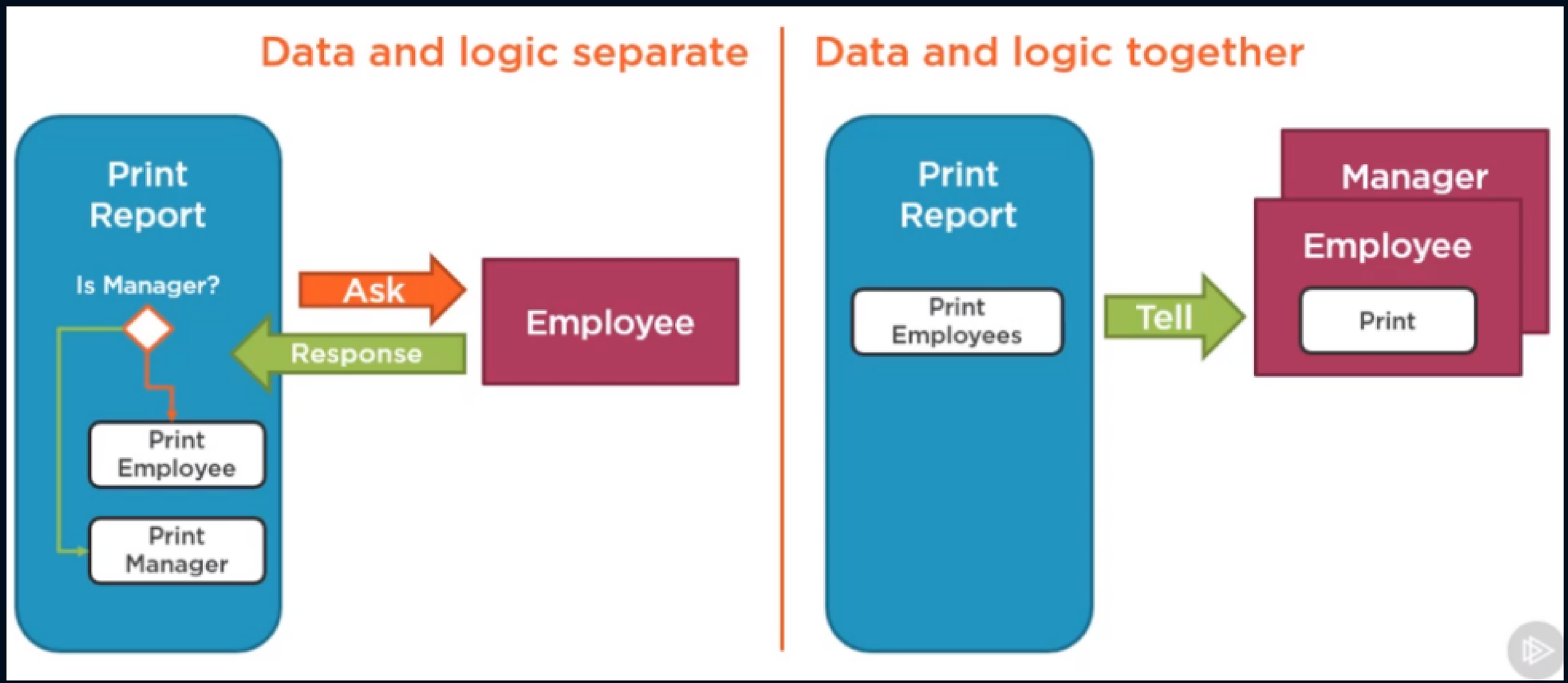
```
public class DoAnything<TArg, TResult>
{
    private Func<TArg, TResult> _function;
    public DoAnything(Func<TArg, TResult> function)
    {
        _function = function;
    }
    public TResult Execute(TArg a)
    {
        return _function(a);
    }
}
```

Extremely Concrete

```
public class DoSomethingElse
{
    public void SomethingElse()
    {
        var doThing = new DoOneThing();
        doThing.Execute();
        // other stuff
    }
}
```

"new" is glue

LSP: Tell don't ask



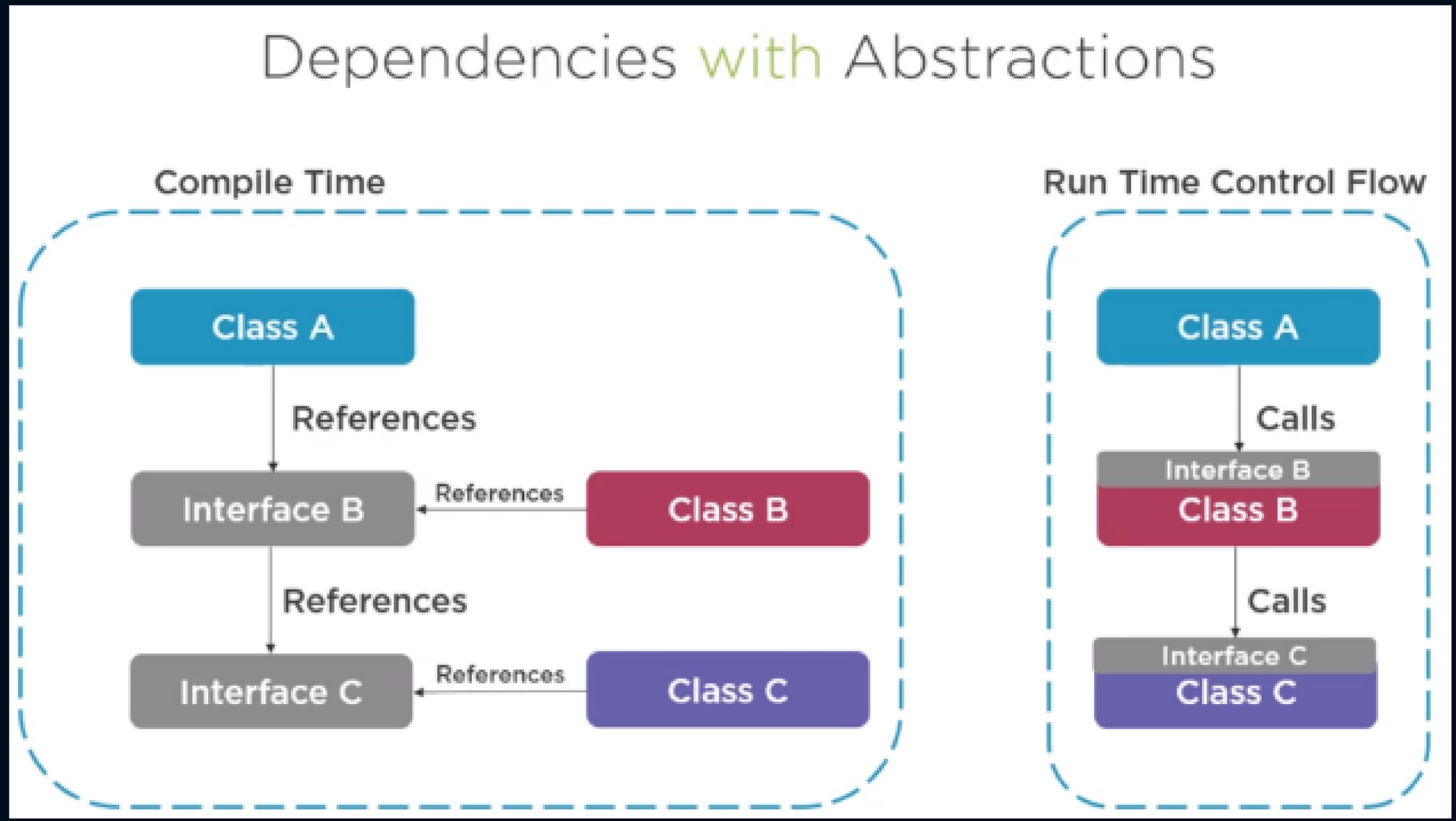
ISP: Example

```
public interface INotificationService
{
    void SendText(string SmsNumber, string message);

    void SendEmail(string to, string from,
                   string subject, string body);
}
```

Split It Up

```
public interface IEmailNotificationService
{
    void SendEmail(string to, string from,
                   string subject, string body);
}

public interface ITextNotificationService
{
    void SendText(string SmsNumber, string message);
}
```

DIP: Depending on Abstractions



DIP: Example

Depending on Details

```
public interface IOrderDataAccess
{
    SqlDataReader ListOrders(SqlParameterCollection params);
}
```

Abstractions Should Not Depend on Details

```
public interface IOrderDataAccess
{
    List<Order> ListOrders(Dictionary<string,string> params);
}
```