SOLID

SOLID

Initial	Concept
S	Single responsibility principle – a class should have only a single responsibility (i.e. only one potential change in the software's specification should be able to affect the specification of the class)
O	Open/closed principle – “software entities … should be open for extension, but closed for modification”
L	Liskov substitution principle – “objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program”
I	Interface segregation principle – “many client-specific interfaces are better than one general-purpose interface”
D	Dependency inversion principle – one should “Depend upon Abstractions. Do not depend upon concretions”

OOD Principals. What that?

Recipes, best practices how to write

- Clear, easy understand code
- Maintainable (flexible, expandable) code

SRP: Single Responsibility Principle

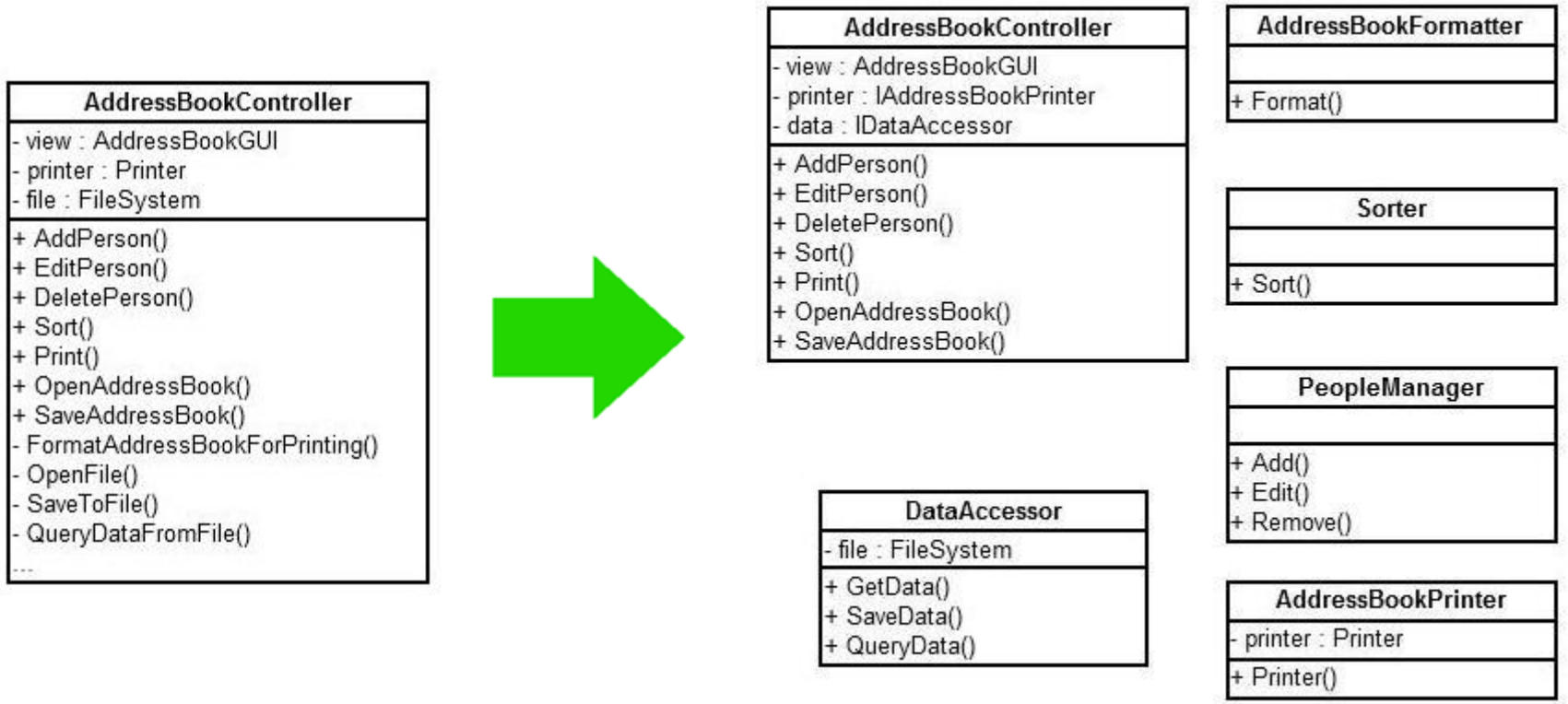
SRP: Single Responsibility Principle

a class or module should have one, and only one, reason to change

what is unnecessary here?

```
public class UserAccount
{
    public string AccountNumber { get; set; }
    public decimal CurrentBallance { get; set; }
    public string Email { get; set; }
    public string Phone { get; set; }
    public string Role { get; set; }

    public TaxTable CalculateTaxes(int year)
    {
        //...
    }
    public bool ApplyTaxes(TaxTable tax)
    {
        //...
    }
}
```

SRP: Single Responsibility Principle



Open/Closed Principle

Software entities should be open for extension, but closed for modification

Open/Closed Principle

- How can we do this?
  - Abstraction
  - Polymorphism
  - Inheritance
  - Composition
  - Proxy implementation
  - Decoration

Open/Closed Principle

"*software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification*“

Initial version:

```
class LogViewer
{
    public IEnumerable<Transaction> GetByDate(DateTime dateTime){ ... }
}
```

With changed requirements:

```
class LogViewer
{
    public IEnumerable<Transaction> GetByDate(DateTime dateTime){ ... }
    public IEnumerable<Transaction> GetByUser(string name){ ... }
    public IEnumerable<Transaction> GetByDateAndUser(DateTime dateTime, string name){ ... }
}
```

OCP:

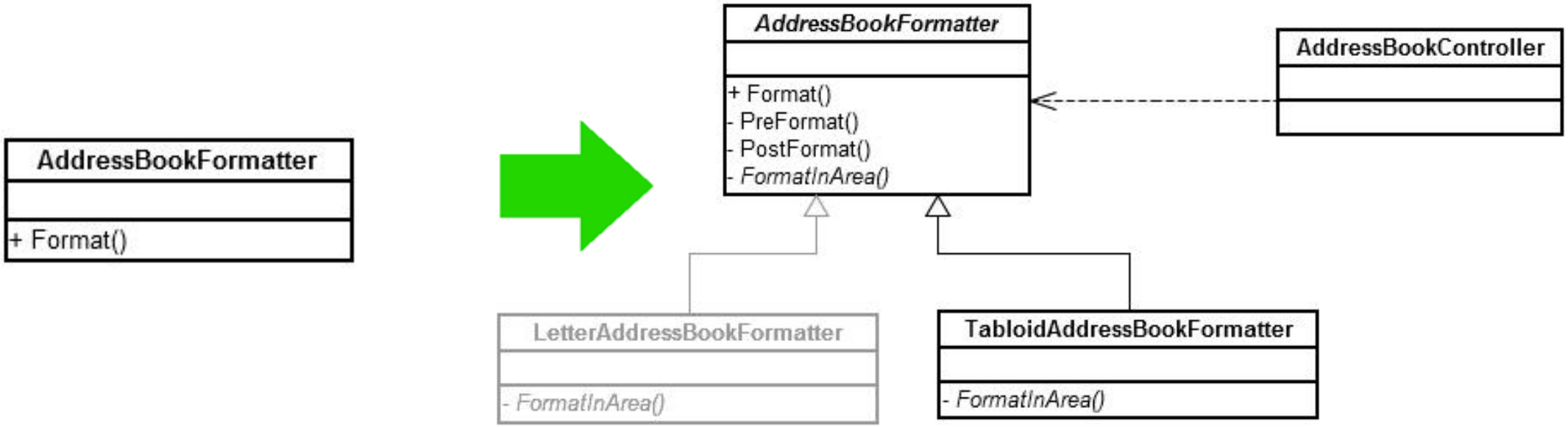
```
class LogViewer
{
    public IEnumerable<Transaction> GetTransaction(GetSpecification spec){ ... }
}
```

Liskov substitution principle

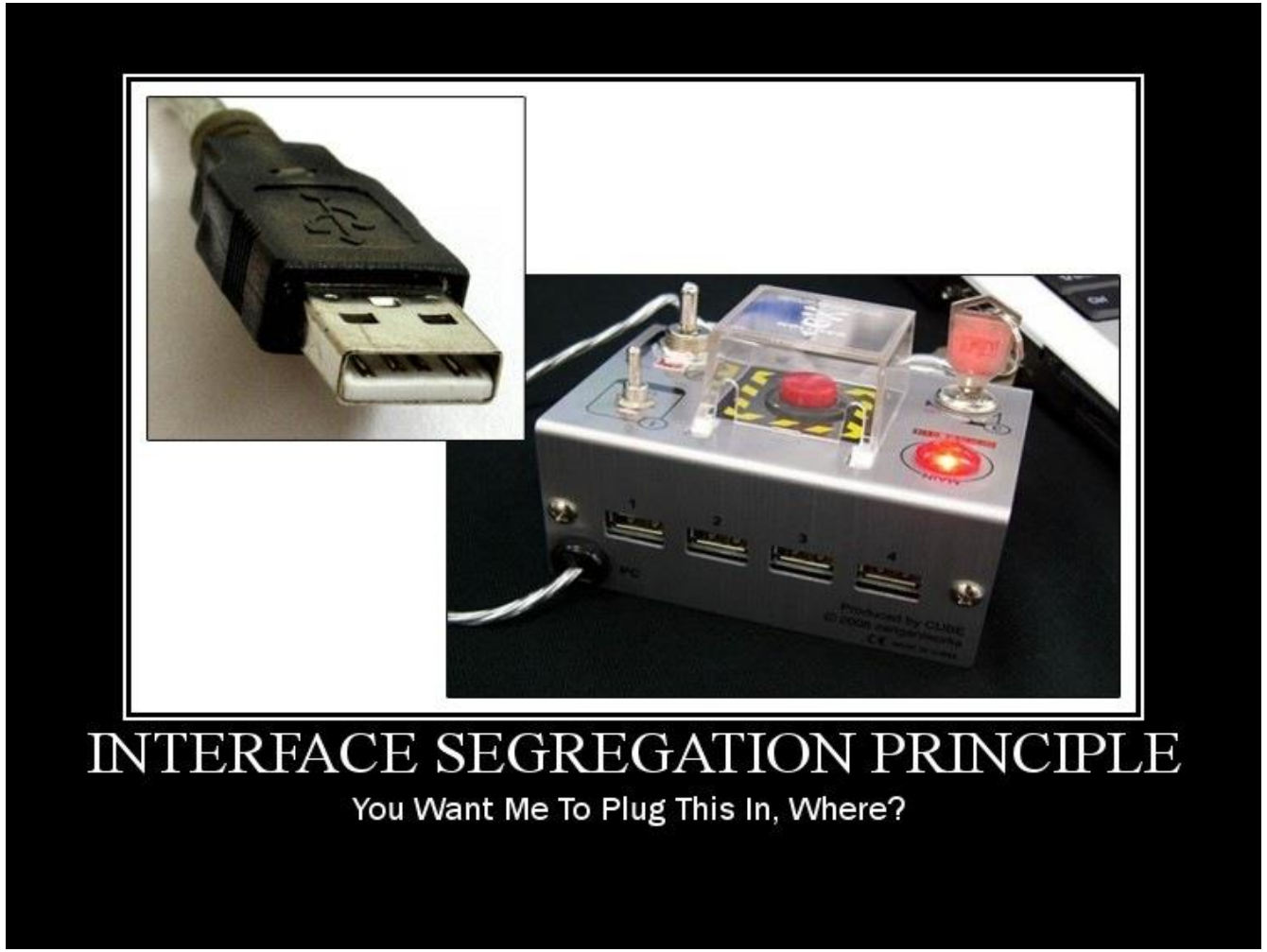
Liskov substitution principle

Clients that use references to base classes must be able to use objects of derived classes without knowing it.

Liskov substitution principle



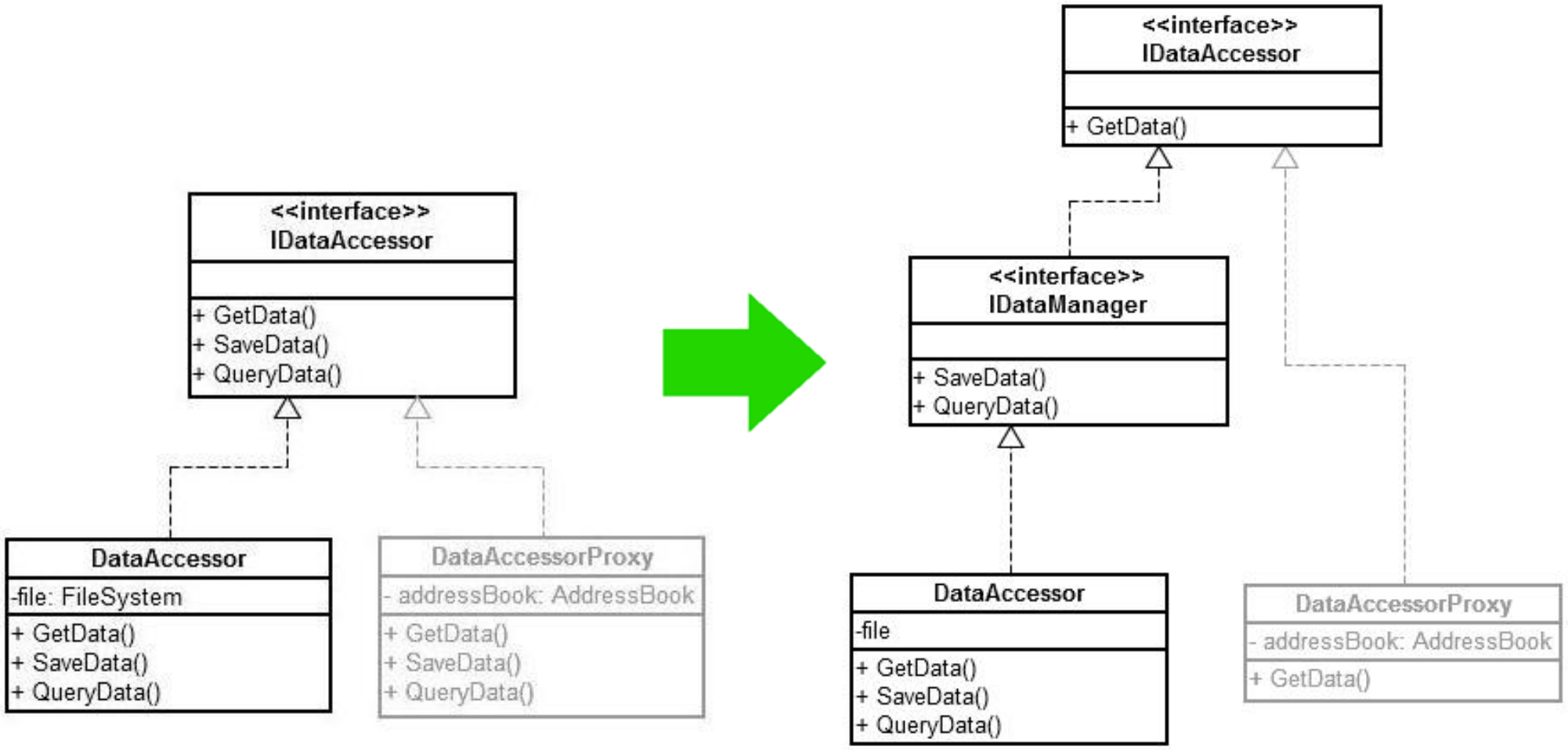
Interface Segregation Principle



Interface Segregation Principle

Clients should not be focused to depend upon interfaces that they do not used

Interface Segregation Principle



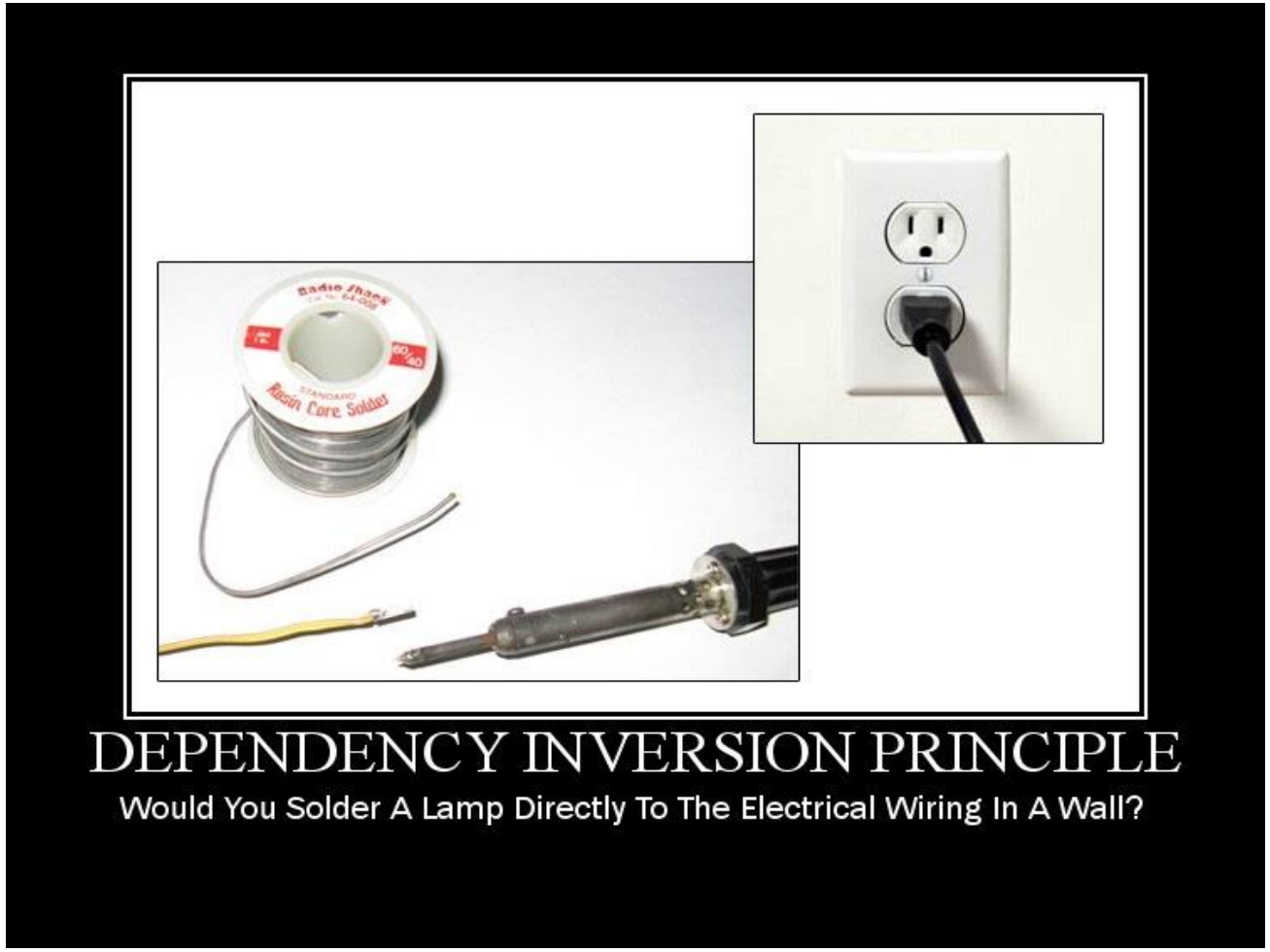
Interface Segregation Principle

no client should be forced to depend on methods it does not use

MembershipProvider for ASP.NET is a good example of huge interfaces

```
abstract class ServiceClient
{
    public string ServiceUri{ get; set; }
    public abstract void SendData(object data);
    public abstract void Flush();
}

class HttpServiceClient : ServiceClient
{
    public override void SendData(object data)
    {
        var channel = OpenChannel(ServiceUri);
        channel.Send(data);
    }

    public override void Flush()
    {
        // DO NOTHING
    }
}

class BufferingHttpServiceClient : ServiceClient
{
    public override void SendData(object data)
    {
        Buffer.Write(data);
    }

    public override void Flush()
    {
        var channel = OpenChannel(ServiceUri);
        channel.Send(Buffer.GetAll());
    }
}
```

Dependency Inversion Principle



Dependency Inversion Principle

High-level modules should not depend on low-level modules. Both should depend on abstractions

Abstractions should not depend upon details. Details should depend upon abstractions

Dependency Inversion Principle

Dependency Inversion Principle

A. *High-level modules should not depend on low-level modules. Both should depend on abstractions.*

B. *Abstractions should not depend upon details. Details should depend upon abstractions.*

```
public class OrderProcessor
{
    private readonly IDiscountCalculator _discountCalculator;
    private readonly ITaxStrategy _taxStrategy;

    public OrderProcessor(IDiscountCalculator discountCalculator,
                          ITaxStrategy taxStrategy)
    {
        _taxStrategy = taxStrategy;
        _discountCalculator = discountCalculator;
    }

    public decimal CalculateTotal(Order order)
    {
        decimal itemTotal = order.GetItemTotal();
        decimal discountAmount = _discountCalculator.CalculateDiscount(order);

        decimal taxAmount = _taxStrategy.FindTaxAmount(order);

        decimal total = itemTotal - discountAmount + taxAmount;

        return total;
    }
}
```