Innovativ Days 2018

Smarter Together

SOLID architecture principles

KHALID BOURZAYQ

Innovativ Days
Smarter Together

SOMMAIRE

- Introduction
- C’est quoi SOLID
- SRP (Single Responsability principal)
- OCP (Open closed principal)
- LSP (Liskov substitution principal)
- ISP (Interface segragation principal)
- DIP (Dependency inversion principal)

Innovativ Days
Smarter Together

Any fool can write code that a computer can understand. Good programmers write code that humans can understand

M. Fowler : https://en.wikipedia.org/wiki/Martin_Fowler

Innovativ Days
Smarter Together

C’est quoi SOLID

- **SOLID** sont cinq principes fondamentaux qui aident à créer une bonne architecture logicielle.
- Induit par R**obert Cecil Martin (Uncle BOB).**

Innovativ Days
Smarter Together

C’est quoi SOLID

- Un acronyme d'acronymes.
- Une collection de bonnes pratiques.
  - Faible couplage
  - Maintenabilité
- C’est des principes – **Ce n’est pas des règles.**
- Se focalise sur la gestion de dépendance

Innovativ Days
Smarter Together

C’est quoi SOLID

- Gestion de dépendance
  - Une faible gestion de dépendance
    - Difficile à changer / Tester
    - Fragile
    - Non-réutilisable
  - Une bonne gestion de dépendance
    - Flexible
    - Robuste
    - Réutilisable

Innovativ Days
Smarter Together

SRP

A class should have only one job

Innovativ Days
Smarter Together

"S" - SRP (Single responsibility principle)

- La meilleur façon de comprendre SOLID et de comprendre le problème qu'il essaye de résoudre.
- Jetons un coup d'œil sur le code ci-dessous et détectant le problème

```
7  namespace SolidApp.NonSolidClasses
8  {
9      public class Customer
10     {
11         public void Add()
12         {
13             try
14             {
15                 // Data access logic here.
16             }
17             catch (Exception ex)
18             {
19                 System.IO.File.WriteAllText(@"d:\Error.txt", ex.ToString());
20             }
21         }
22     }
23 }
24
```

"S" - SRP (Single responsibility principle)

- La classe client fait des traitements **QU'ELLE N'EST PAS SUPPOSÉ À FAIRE.**

"S" - SRP (Single responsibility principle)

- SRP : Une classe ne devrait avoir qu'une seule responsabilité
- En appliquant SRP sur notre cas, on va déplacer la fonctionnalité de log dans une nouvelle classe.

```
namespace SolidApp.SolidClasses
{
    public class FileLogger
    {
        public void Handle(string error)
        {
            System.IO.File.WriteAllText(@"d:\Error.txt", error);
        }
    }
}
```

```
public class Customer
{
    private FileLogger obj = new FileLogger();
    public void Add()
    {
        try
        {
            // Data access logic here.
        }
        catch (Exception ex)
        {
            obj.Handle(ex.ToString());
        }
    }
}
```

Innovativ Days
Smarter Together

OCP

Objects or entities should be open for extension, but closed for modification

Innovativ Days
Smarter Together

"O" - OCP Open Closed Principle

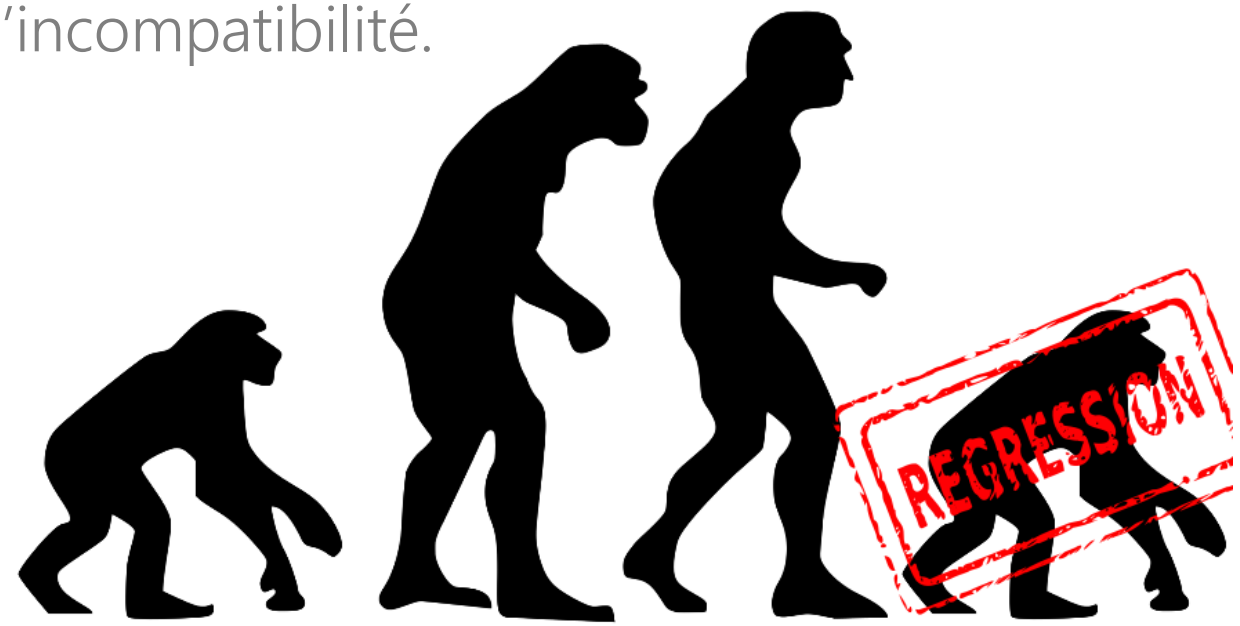
- On continue avec la même classe client et y ajoutant une nouvelle propriété qui représente le type du client et une méthode qui calcule la remise en fonction du type du client.

```
9   public class Customer
10  {
11
12      private int _customerType;
13
14      public int CustomerType
15      {
16          get { return _customerType; }
17          set { _customerType = value; }
18      }
19
20      public double GetDiscount(double totalSales)
21      {
22          if (_customerType == 1)
23          {
24              return totalSales - 100;
25          }
26          return totalSales - 50;
27      }
28
```

Innovativ Days
Smarter Together

“O” - OCP Open Closed Principle

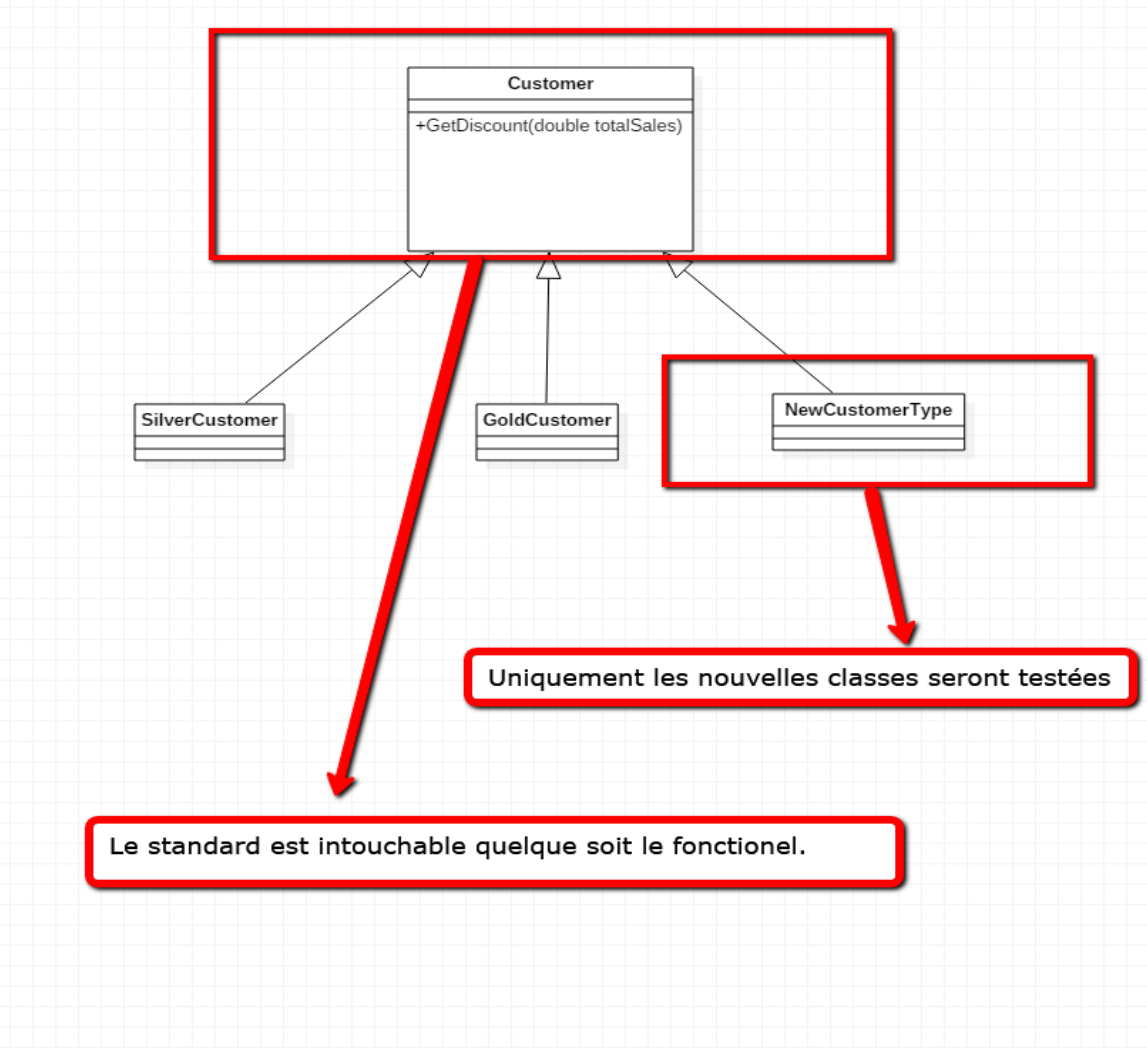
- Problèmes :
  - Autant de nouvelle classes, autant de clause if.
  - Possibilité de régression.
  - Possibilité d’incompatibilité.



Innovativ Days
Smarter Together

"O" - OCP Open Closed Principle

- Solution



Innovativ Days
Smarter Together

"O" - OCP Open Closed Principle

```
public class Customer
{
    private FileLogger obj = new FileLogger();
    public void Add()
    {
        try
        {
            // Data access logic here.
        }
        catch (Exception ex)
        {
            obj.Handle(ex.ToString());
        }
    }

    public virtual double GetDiscount(double totalSales)
    {
        return totalSales;
    }
}

internal class SilverCustomer : Customer
{
    public override double GetDiscount(double totalSales)
    {
        return base.GetDiscount(totalSales) - 50;
    }
}

internal class GoldCustomer : SilverCustomer
{
    public override double GetDiscount(double totalSales)
    {
        return base.GetDiscount(totalSales) - 100;
    }
}
```

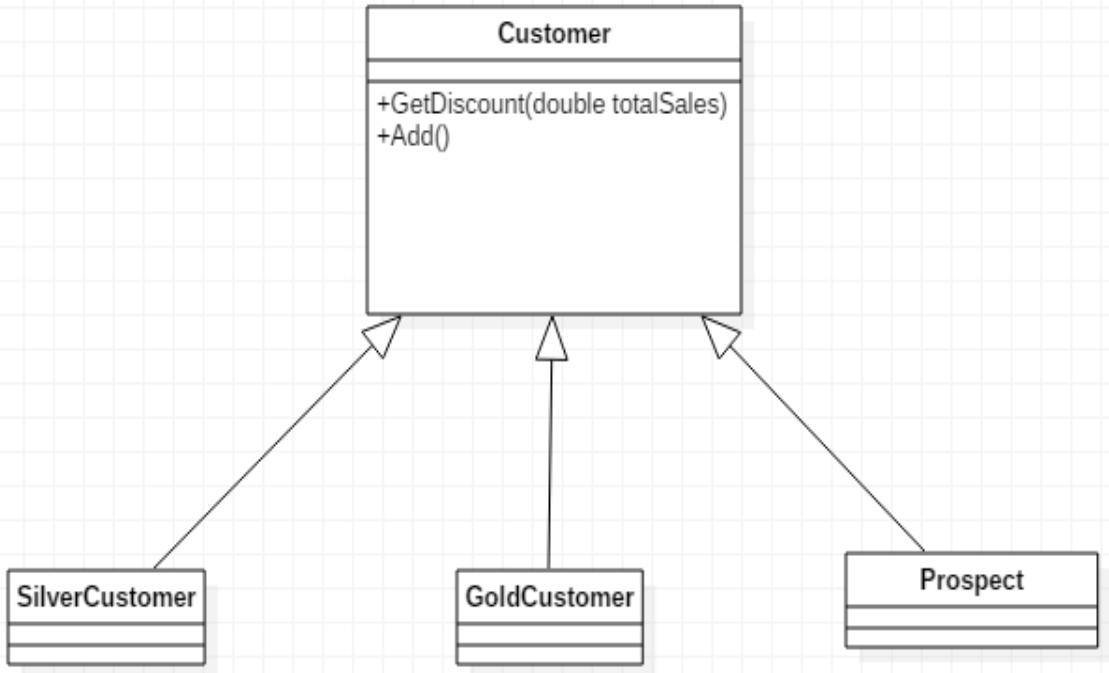
Innovativ Days
Smarter Together

LSP

Subtypes must be substitutable for their base types.

Innovativ Days
Smarter Together

“L” - LSP Liskov substitution Principle



Innovativ Days
Smarter Together

“L” - LSP Liskov substitution Principle

```
public class Prospect : Customer
{
    public override double GetDiscount(double totalSales)
    {
        return totalSales - 5;
    }

    public override void Add()
    {
        throw new NotSupportedException();
    }
}
```

Innovativ Days
Smarter Together

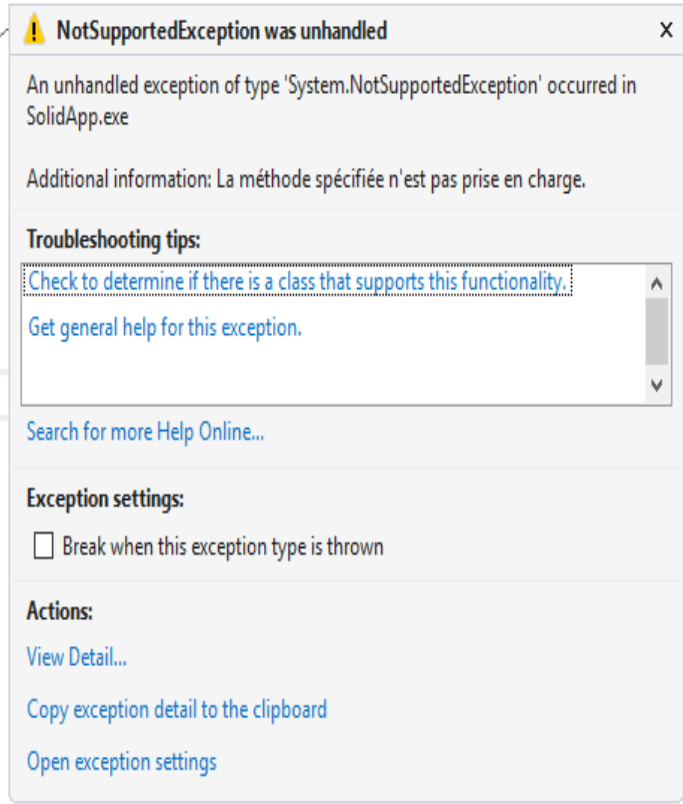
“L” - LSP Liskov substitution Principle

```
var customers = new List<Customer>
{
    new SilverCustomer(), new GoldCustomer(), new Prospect()
};

foreach (var o in customers)
{
    o.Add();
}
```

Innovativ Days
Smarter Together

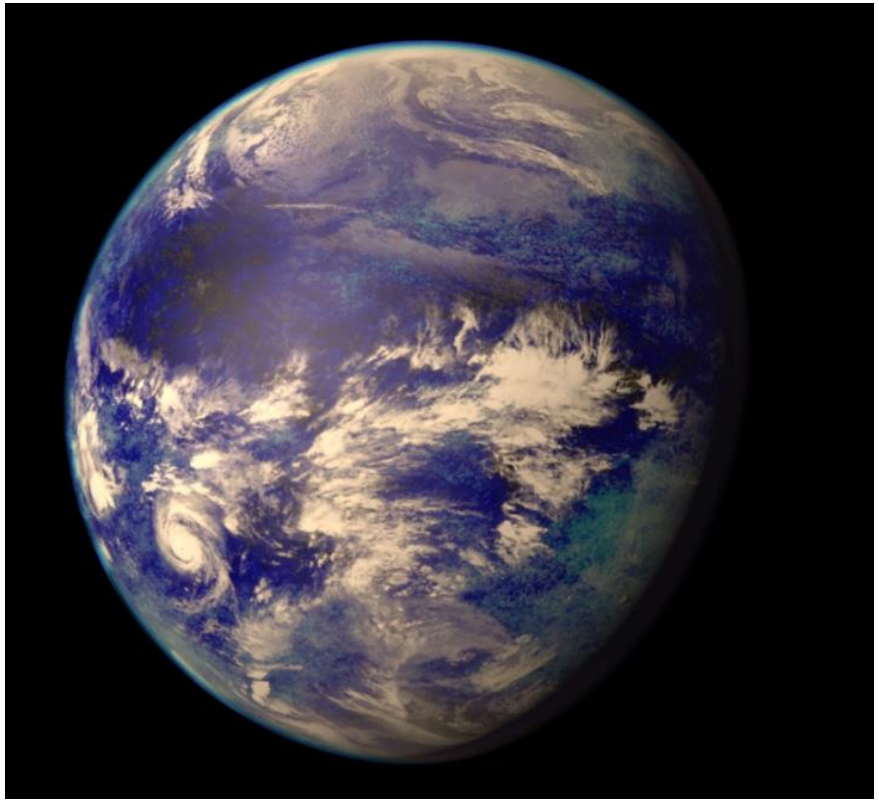
“L” - LSP Liskov substitution Principle

```
public class Prospect : SolidClasses.Customer
{
    public override double GetDiscount(double totalSales)
    {
        return totalSales - 5;
    }

    public override void Add()
    {
        throw new NotSupportedException();
    }
}
```



Innovativ Days
Smarter Together

“L” - LSP Liskov substitution Principle



Kepler 186f

La terre

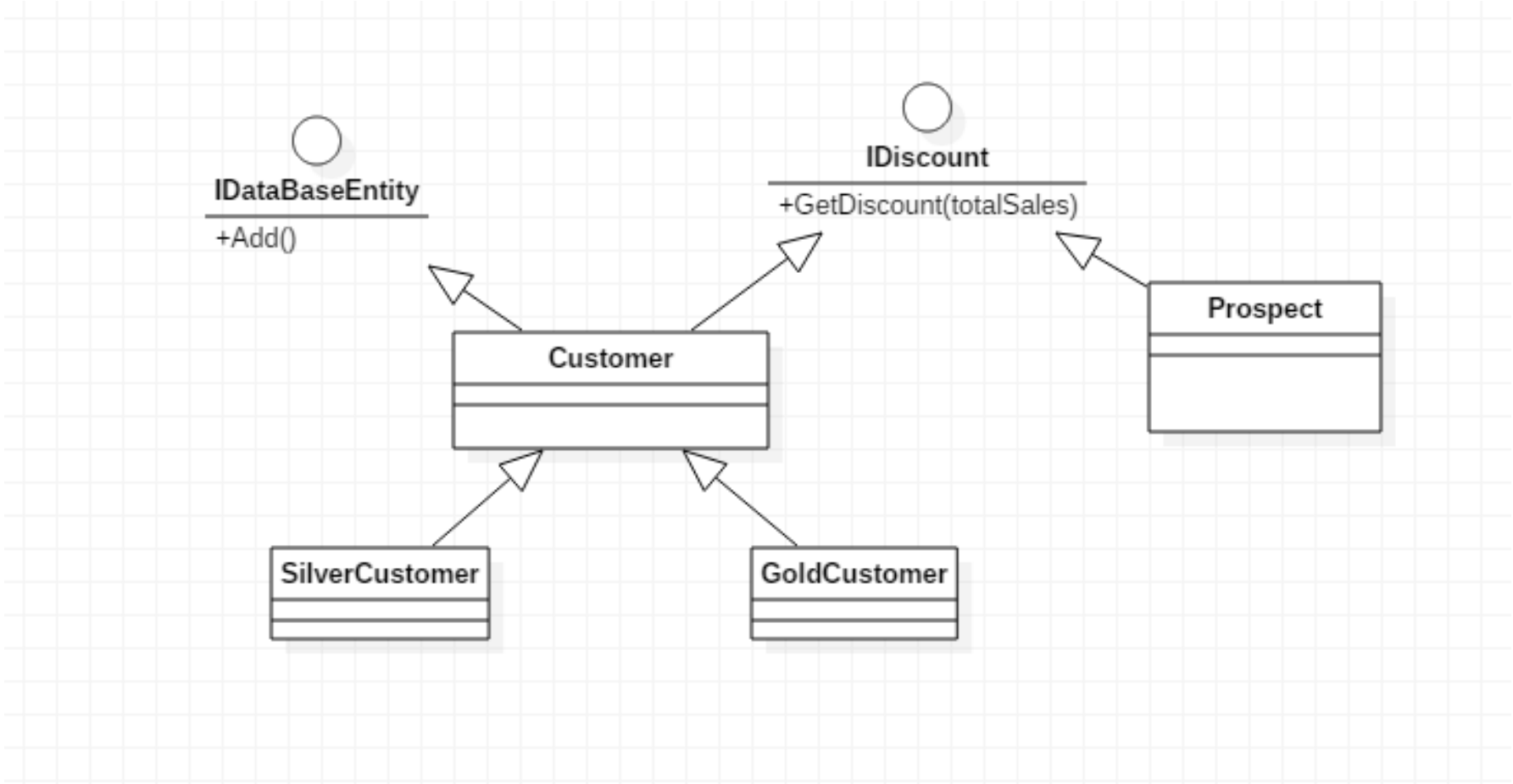
Innovativ Days
Smarter Together

LSP

*Si **q(x)** est une propriété démontrable pour tout objet **x** de type **T**, alors **q(y)** est vraie pour tout objet **y** de type **S** tel que **S** est un sous-type de **T***

Innovativ Days
Smarter Together

"L" - LSP Liskov substitution Principle



Innovativ Days
Smarter Together

"L" - LSP Liskov substitution Principle

```
public abstract class Customer : IDiscount, IDataBaseEntity
{
    private FileLogger obj = new FileLogger();
    public void Add()
    {
        try
        {
            //Data access logic here.
        }
        catch (Exception ex)
        {
            obj.Handle(ex.ToString());
        }
    }

    public abstract double GetDiscount(double totalSales);
}

public class SilverCustomer : Customer
{
    public override double GetDiscount(double totalSales)
    {
        return totalSales - 50;
    }
}

public class GoldCustomer : SilverCustomer
{
    public override double GetDiscount(double totalSales)
    {
        return totalSales - 100;
    }
}
```

```
public interface IDataBaseEntity
{
    void Add();
}
```

```
public interface IDiscount
{
    double GetDiscount(double totalSales);
}
```

```
public class Prospect : IDiscount
{
    public double GetDiscount(double totalSales)
    {
        return totalSales;
    }
}
```

```
var customers = new List<IDataBaseEntity>
{
    new SilverCustomer(), new GoldCustomer(), new Prospect()
};

foreach (var o in customers)
{
    o.Add();
}
```

Innovativ Days
Smarter Together

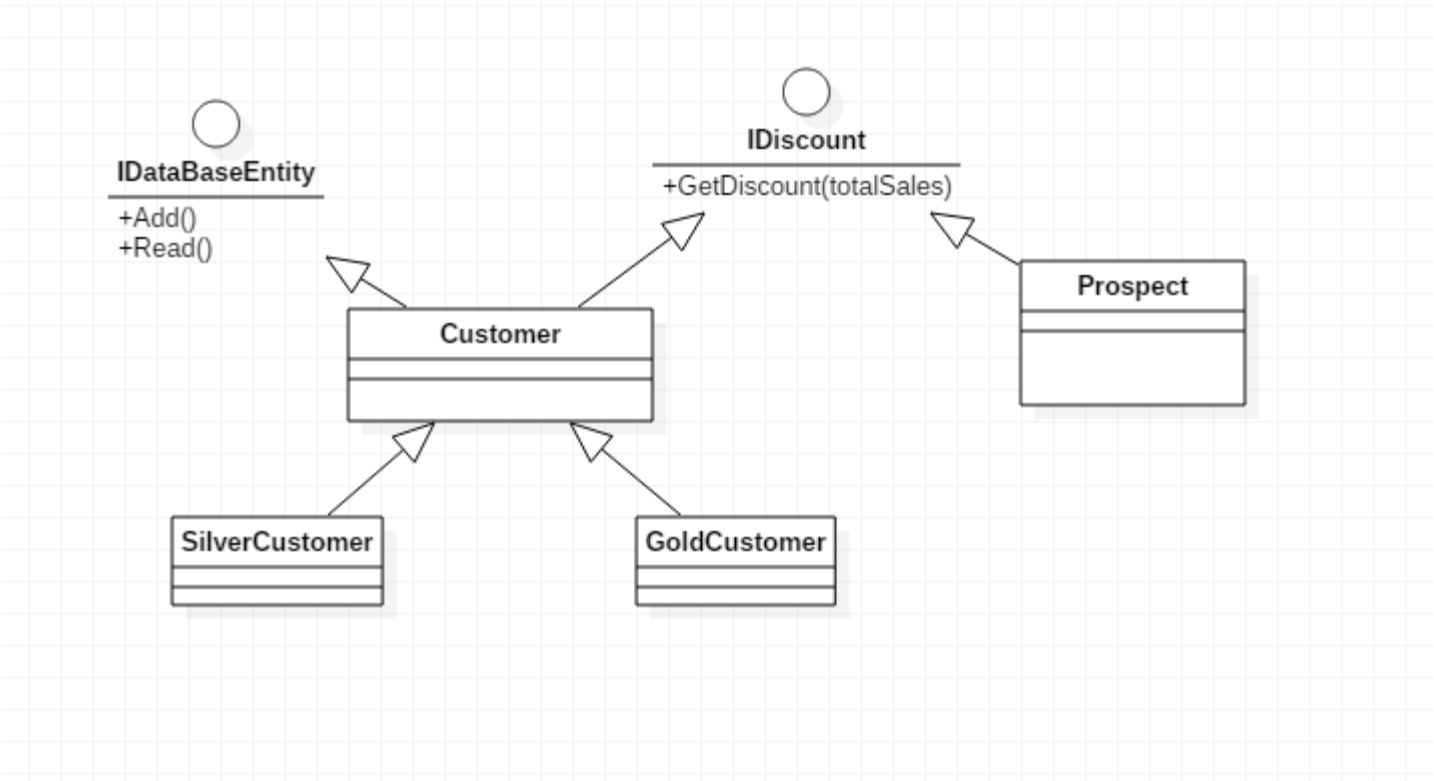
ISP

A client should never be forced to implement an interface that it doesn't use or clients shouldn't be forced to depend on methods they do not use.

Innovativ Days
Smarter Together

"I" - ISP Inteface segregation Principle

- Supposant qu'il y a des nouveaux clients qui nous demandent une nouvelle méthode pour la lecture des données. Les autres ne sont pas intéressés



Innovativ Days
Smarter Together

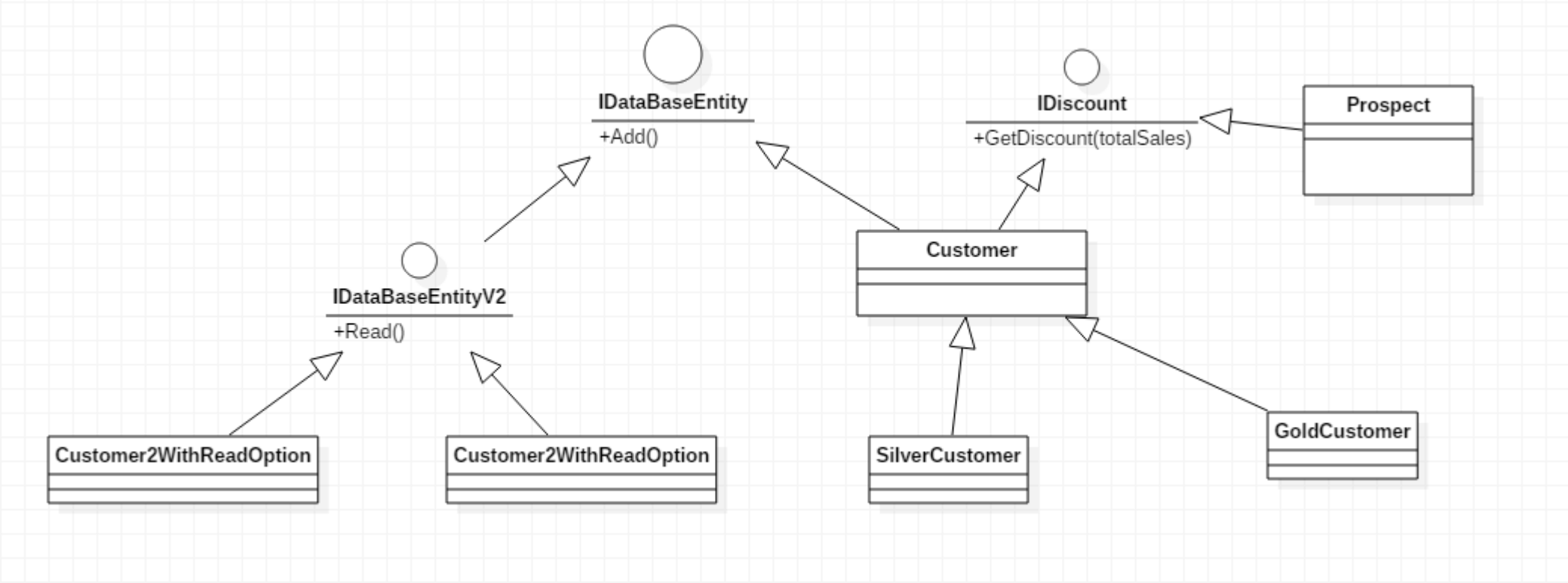
“I” - ISP Inteface segregation Principle

- En faisant ce genre de conception rapide, nous avons perturbé tous les clients utilisant notre interface IDataBaseEntity et on va les forcer à implémenter une méthode dont ils n’ont pas besoin.

Innovativ Days
Smarter Together

"I" - ISP Inteface segregation Principle

- La solution consiste à séparer la nouvelle fonctionnalité dans une nouvelle interface qui va répondre au besoin des nouveaux clients



Innovativ Days
Smarter Together

DIP

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

Innovativ Days
Smarter Together

"D" - DIP Dependency inversion principle

```
public abstract class Customer : IDiscount, IDataBaseEntity
{
    private FileLogger obj = new FileLogger();
    public void Add()
    {
        try
        {
            //Data access logic here.
        }
        catch (Exception ex)
        {
            obj.Handle(ex.ToString());
        }
    }

    public abstract double GetDiscount(double totalSales);
}

public class SilverCustomer : Customer
{
    public override double GetDiscount(double totalSales)
    {
        return totalSales - 50;
    }
}

public class GoldCustomer : SilverCustomer
{
    public override double GetDiscount(double totalSales)
    {
        return totalSales - 100;
    }
}
```

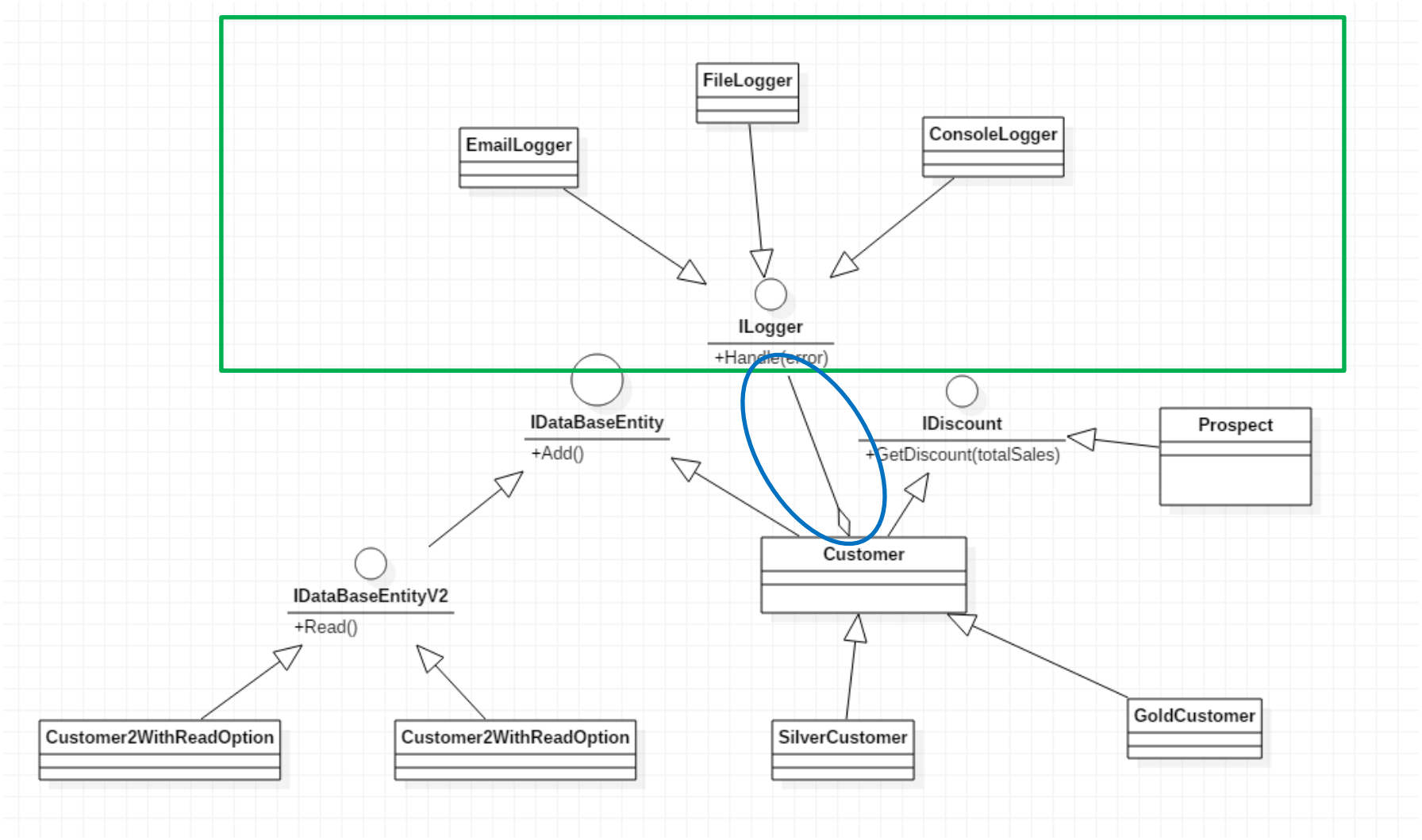
Innovativ Days
Smarter Together

"D" - DIP Dependency inversion principle

- On a deux gros problèmes dans notre exemple:
  - La dépendance forte à l'objet FileLogger.
  - La classe client prend la responsabilité de création d'une dépendance.
- La solution est de déléguer / inverser cette responsabilité à une autre classe et cela va résoudre notre problème.
- On aura aussi la possibilité de switcher de type de logger quand on veut vu qu'on ne va plus dépendre d'un objet concret

Innovativ Days
Smarter Together

"D" - DIP Dependency inversion principle



Innovativ Days
Smarter Together

"D" - DIP Dependency inversion principle

```
public abstract class Customer : IDiscount, IDataBaseEntity
{
    private readonly ILogger _obj;

    protected Customer(ILogger logger)
    {
        _obj = logger;
    }
    public void Add()
    {
        try
        {
            //Data access logic here.
        }
        catch (Exception ex)
        {
            _obj.Handle(ex.ToString());
        }
    }

    public abstract double GetDiscount(double totalSales);
}
```

```
public class SilverCustomer : Customer
{
    public override double GetDiscount(double totalSales)
    {
        return totalSales - 50;
    }

    public SilverCustomer(ILogger logger) : base(logger)
    {
    }
}

public class GoldCustomer : SilverCustomer
{
    public override double GetDiscount(double totalSales)
    {
        return totalSales - 100;
    }

    public GoldCustomer(ILogger logger) : base(logger)
    {
    }
}
```

```
ILogger logger= new FileLogger();
List<IDataBaseEntity> customers = new List<IDataBaseEntity>
{
    new SilverCustomer(logger), new GoldCustomer(logger)
};

foreach (IDataBaseEntity o in customers)
{
    o.Add();
}
```

Innovativ Days
Smarter Together

?

Innovativ Days
Smarter Together

Merci