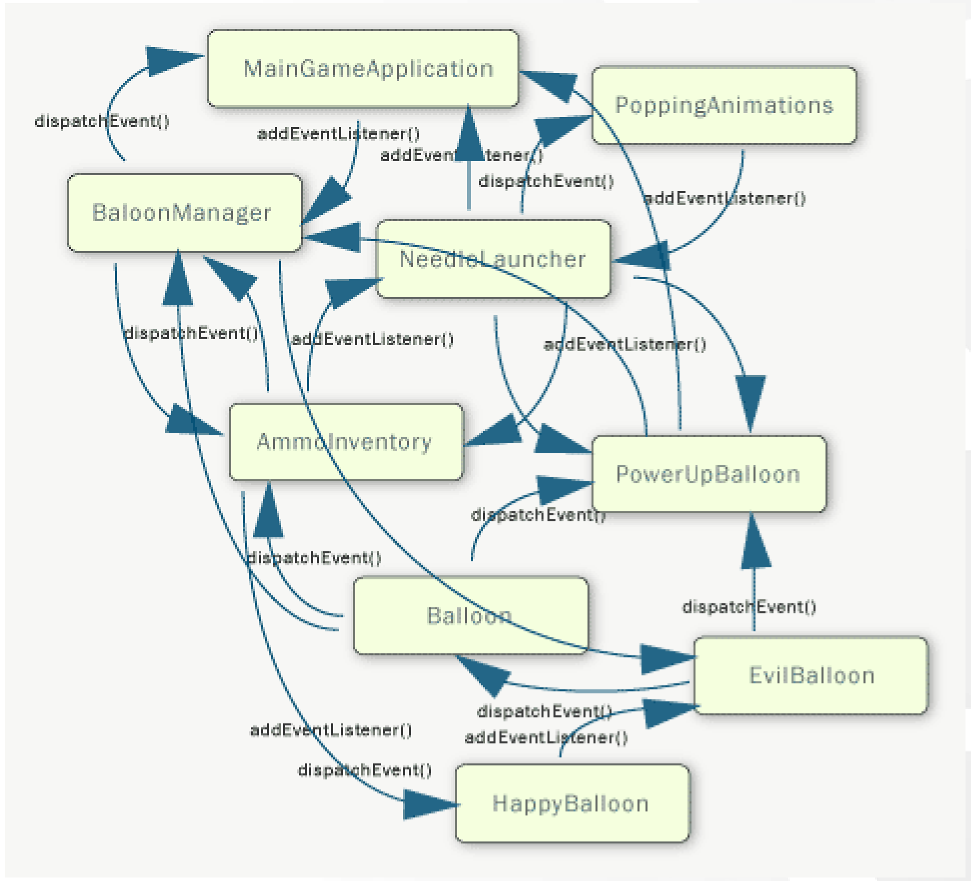
SOLID

S.O.L.I.D Principles

- Made by Robert C. Martin (a.k.a "Uncle Bob")
- Introduced the idea in 1995 while working on several issues on OO Code
- There Was Too much dependency happening !

Symptoms of Bad Code



- Rigidity (קשיחות) – Every change affects many other parts
- Fragility – Things brake in unrelated places
- Immobility – Cannot reuse code outside its original context.

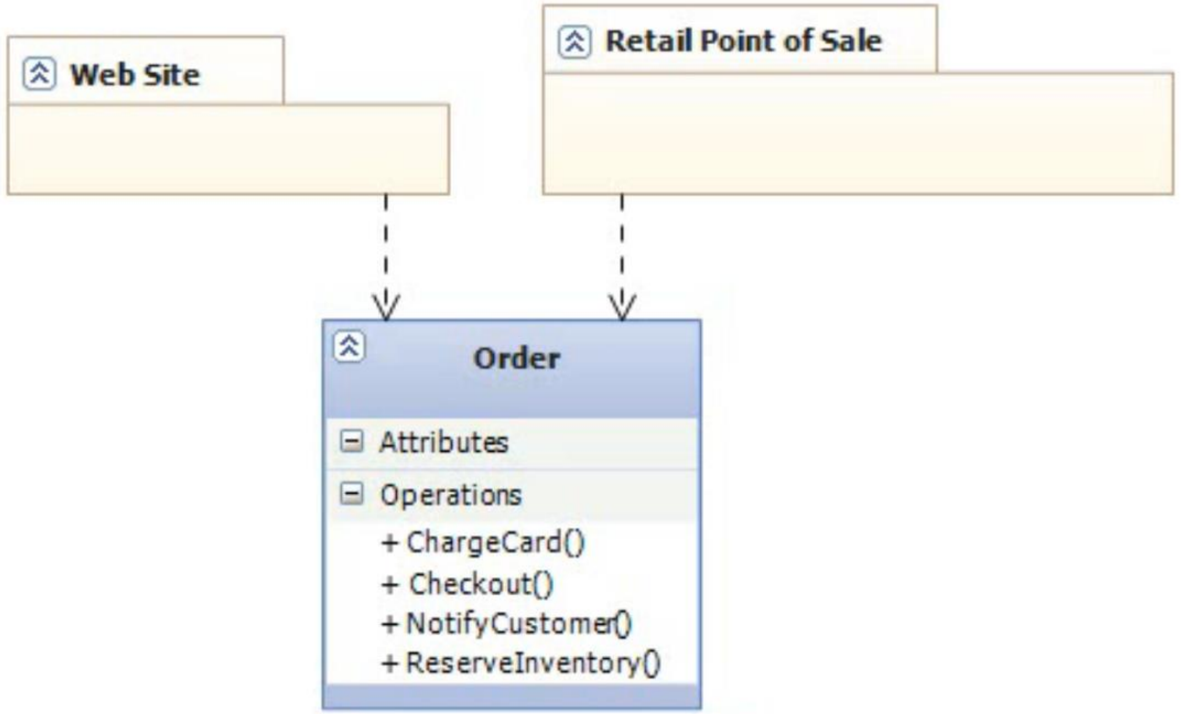
SOLID To The Rescue

- S - Single Responsibility Principle
- O - Open Close Principle
- L - Liskov Substitution Principle
- I - Interface Segregation Principle
- D - Dependency Inversion Principle

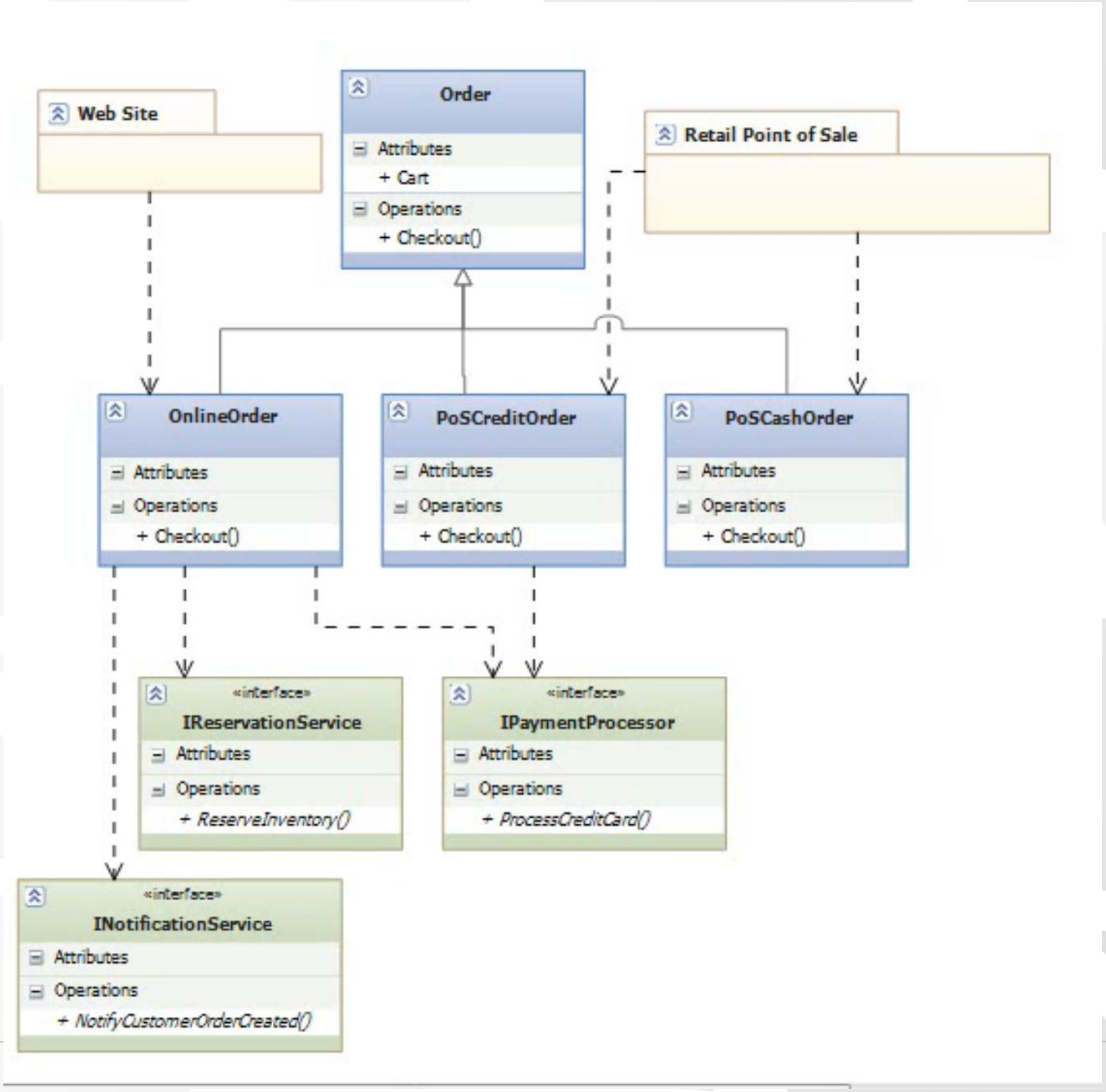
Single Responsibility Pattern (SRP)

A class Should have only 1 reason to change

Problem definition



Refactor



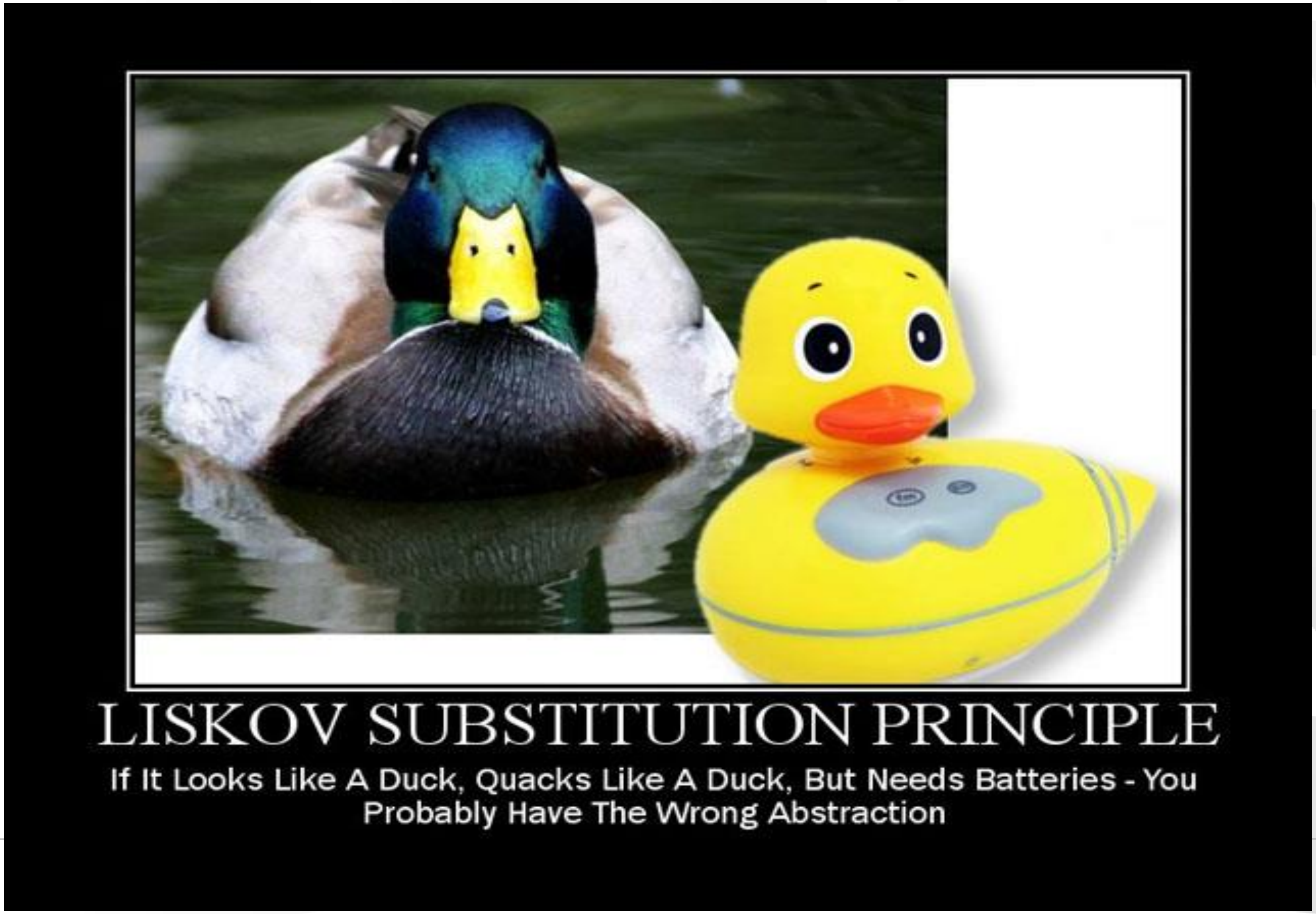
Web Site
Order
Attributes
+ Cart
Operations
+ Checkout()
Retail Point of Sale
OnlineOrder
Attributes
Operations
+ Checkout()
PoSCreditOrder
Attributes
Operations
+ Checkout()
PoSCashOrder
Attributes
Operations
+ Checkout()
«interface»
IReservationService
Attributes
Operations
+ ReserveInventory()
«interface»
IPaymentProcessor
Attributes
Operations
+ ProcessCreditCard()
«interface»
INotificationService
Attributes
Operations
+ NotifyCustomerOrderCreated()

Open Close Principle

Be Open For Extension , Close for Modifications

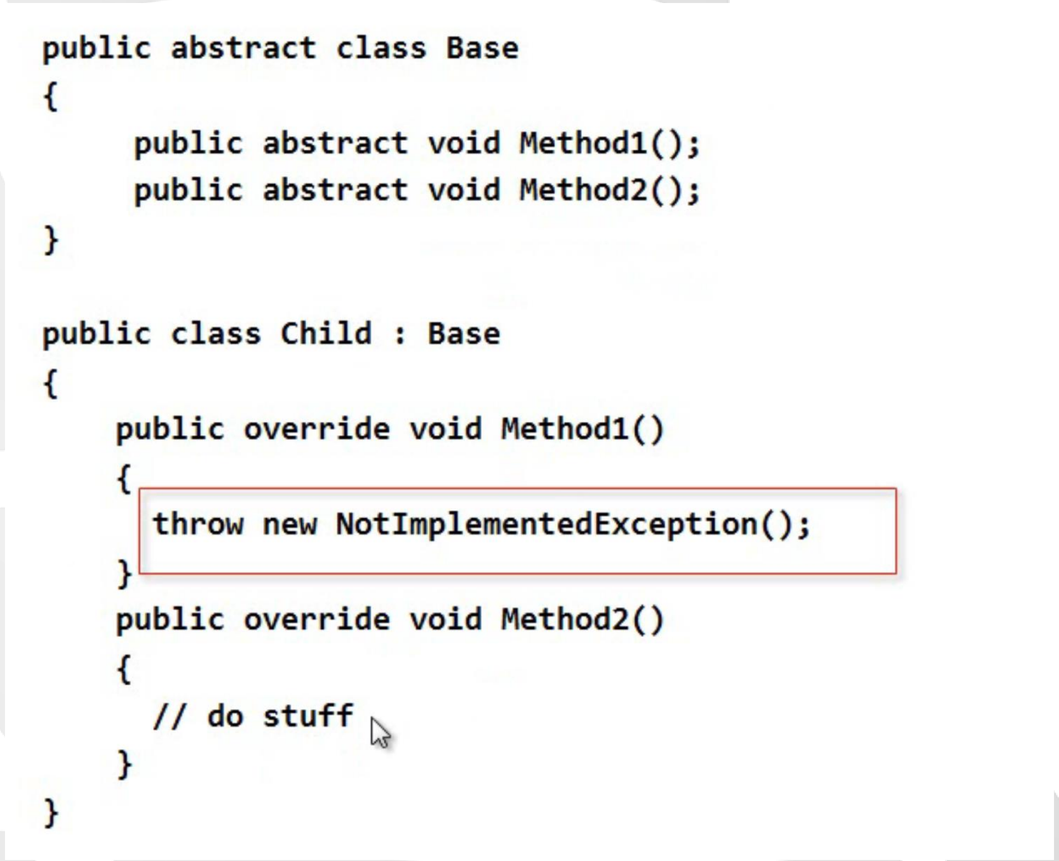
Liskov Substitution

Derived classes can be stand In for Base Classes


LISKOV SUBSTITUTION PRINCIPLE
If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You Probably Have The Wrong Abstraction

LSP Violation “Smell”

```
foreach (var emp in Employees)
{
  if(emp is Manager)
  {
      _printer.PrintManager(emp as Manager);
  }
  else
  {
    _printer.PrintEmployee(emp);
  }
}
```



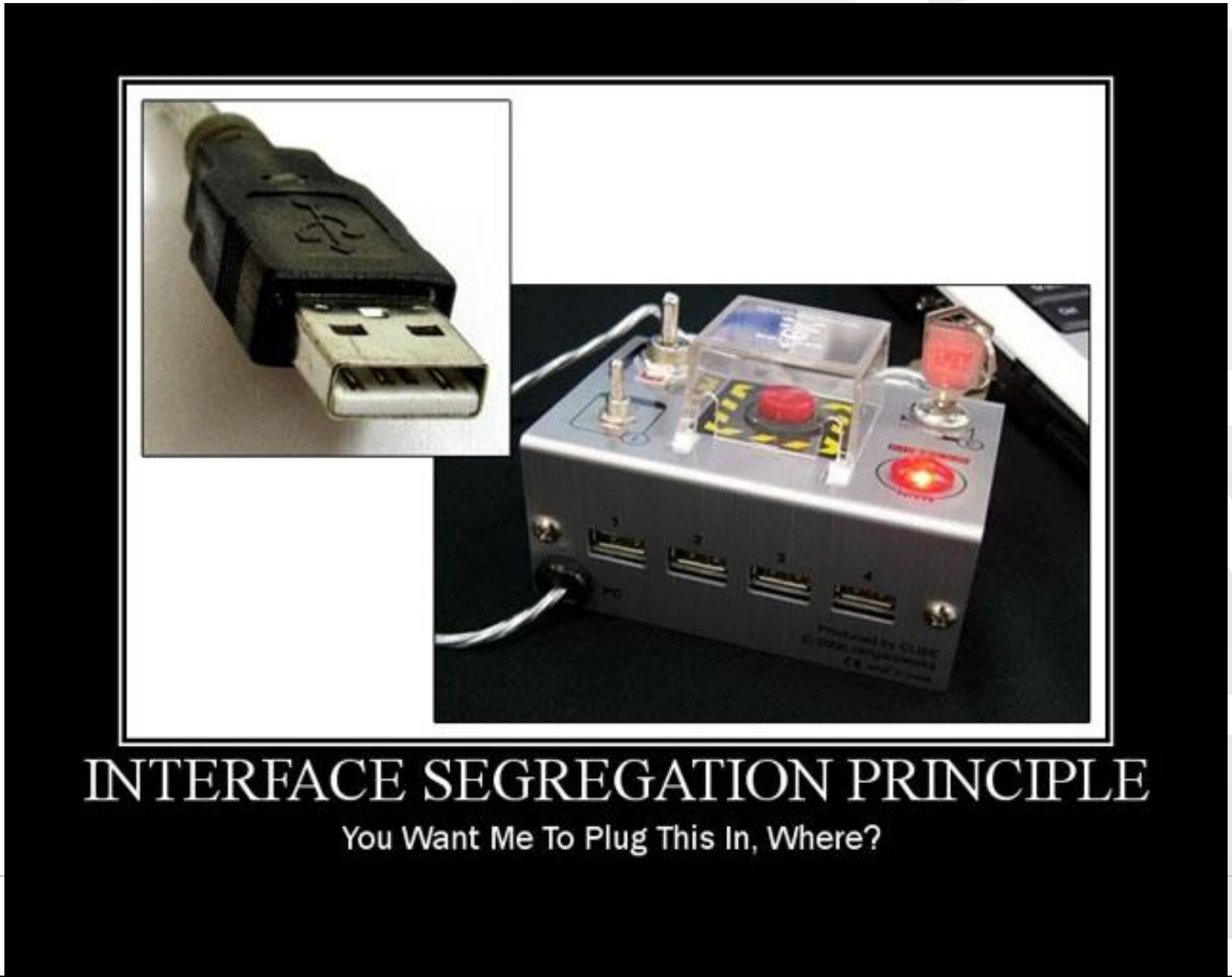
```
public abstract class Base
{
     public abstract void Method1();
     public abstract void Method2();
}

public class Child : Base
{
    public override void Method1()
    {
      throw new NotImplementedException();
    }
    public override void Method2()
    {
      // do stuff
    }
}
```

LSP Tips

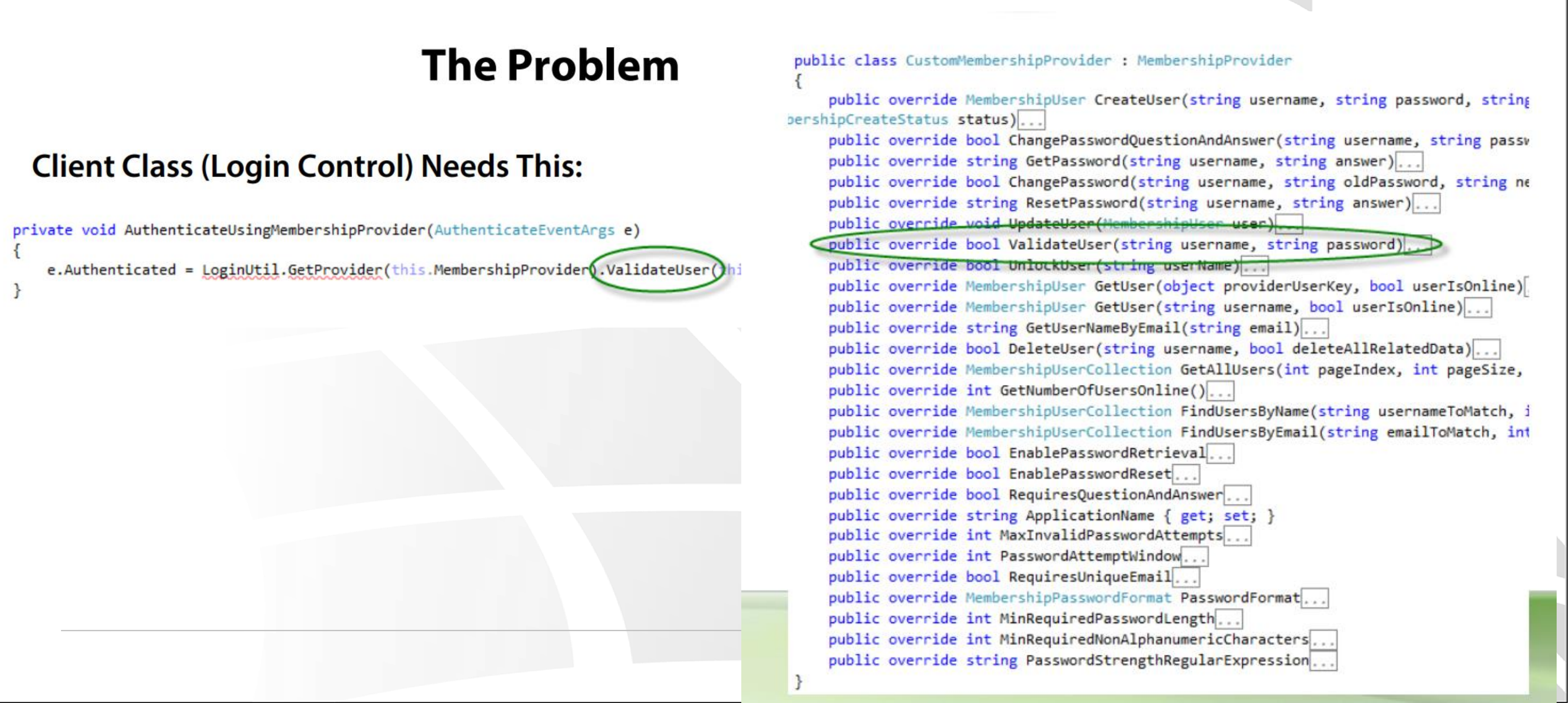
- “Tell Don’t Ask”
  - Don’t interrogate objects for their internal state
  - Tell the object what you want it to do
- Consider refactoring to a new Base Class
  - When you have 2 classes that share behavior but are not subsititutable
  - Create a third class
  - Ensure Substitutability is retained between each class and the base

Interface Segregation Principle

Make fine grained interfaces with specific methods


INTERFACE SEGREGATION PRINCIPLE
You Want Me To Plug This In, Where?

The problem



The Problem

Client Class (Login Control) Needs This:

```
private void AuthenticateUsingMembershipProvider(AuthenticateEventArgs e)
{
    e.Authenticated = LoginUtil.GetProvider(this.MembershipProvider).ValidateUser(thi
}
```

```
public class CustomMembershipProvider : MembershipProvider
{
    public override MembershipUser CreateUser(string username, string password, string
bershipCreateStatus status)...
    public override bool ChangePasswordQuestionAndAnswer(string username, string passw
    public override string GetPassword(string username, string answer)...
    public override bool ChangePassword(string username, string oldPassword, string ne
    public override string ResetPassword(string username, string answer)...
    public override void UpdateUser(MembershipUser user)...
    public override bool ValidateUser(string username, string password)...
    public override bool UnlockUser(string userName)...
    public override MembershipUser GetUser(object providerUserKey, bool userIsOnline)
    public override MembershipUser GetUser(string username, bool userIsOnline)...
    public override string GetUserNameByEmail(string email)...
    public override bool DeleteUser(string username, bool deleteAllRelatedData)...
    public override MembershipUserCollection GetAllUsers(int pageIndex, int pageSize,
    public override int GetNumberOfUsersOnline()...
    public override MembershipUserCollection FindUsersByName(string usernameToMatch, i
    public override MembershipUserCollection FindUsersByEmail(string emailToMatch, int
    public override bool EnablePasswordRetrieval...
    public override bool EnablePasswordReset...
    public override bool RequiresQuestionAndAnswer...
    public override string ApplicationName { get; set; }
    public override int MaxInvalidPasswordAttempts...
    public override int PasswordAttemptWindow...
    public override bool RequiresUniqueEmail...
    public override MembershipPasswordFormat PasswordFormat...
    public override int MinRequiredPasswordLength...
    public override int MinRequiredNonAlphanumericCharacters...
    public override string PasswordStrengthRegularExpression...
}
```

ISP Smells

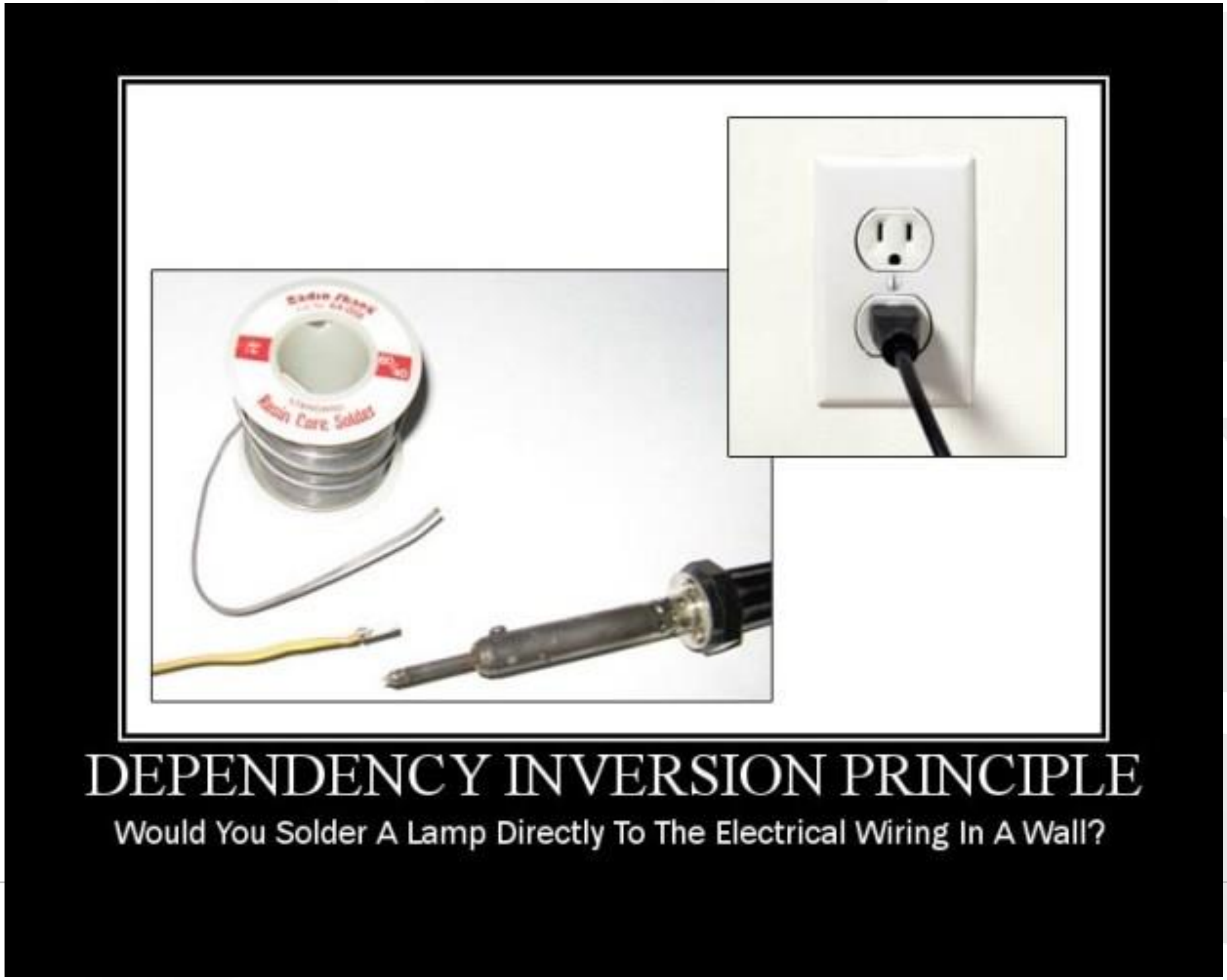
- Unimplemented interface methods

```
public override string resetPassword(string password)
{
  throw new NotImplementedException();
}
```

- Client references a class but only use small portion of it.

Dependency Inversion Principle

Depend On abstraction not on Concrete Implementation


DEPENDENCY INVERSION PRINCIPLE
Would You Solder A Lamp Directly To The Electrical Wiring In A Wall?
Rosin Core Solder

What are Dependencies ?

- Framework
- Third Party Libraries
- Database
- File System
- Email
- Web Services
- System Resources (Clock)
- Configuration
- The new Keyword
- Static methods
- Thread.Sleep
- Random

Class Dependencies – Be Honest !

- Class C'tor should require any dependency the class needs
- Classes whose constructors make this clear have explicit dependency.
- Class that do not : have implicit, hidden dependency .

```
public class HelloWorldHidden
{
    public string Hello(string name)
    {
        if (DateTime.Now.Hour < 12) return "Good morning, " + name;
        if (DateTime.Now.Hour < 18) return "Good afternoon, " + name;
        return "Good evening, " + name;
    }
}
```

Classes should declare what they need !

```
public class HelloWorldExplicit
{
    private readonly DateTime _timeOfGreeting;

    public HelloWorldExplicit(DateTime timeOfGreeting)
    {
        _timeOfGreeting = timeOfGreeting;
    }

    public string Hello(string name)
    {
        if (_timeOfGreeting.Hour < 12) return "Good morning, " + name;
        if (_timeOfGreeting.Hour < 18) return "Good afternoon, " + name;
        return "Good evening, " + name;
    }
}
```

DIP "Smells"

- Use of the "new" keyword

```
foreach(var item in cart.Items)
{
  try
  {
    var inventorySystem = new InventorySystem();
    inventorySystem.Reserve(item.Sku, item.Quantity);
  }
}
```

- Use of static methods/properties

```
message.Subject = "Your order placed on " +
   DateTime.Now.ToString();
```

Or

```
DataAccess.SaveCustomer(myCustomer);
```