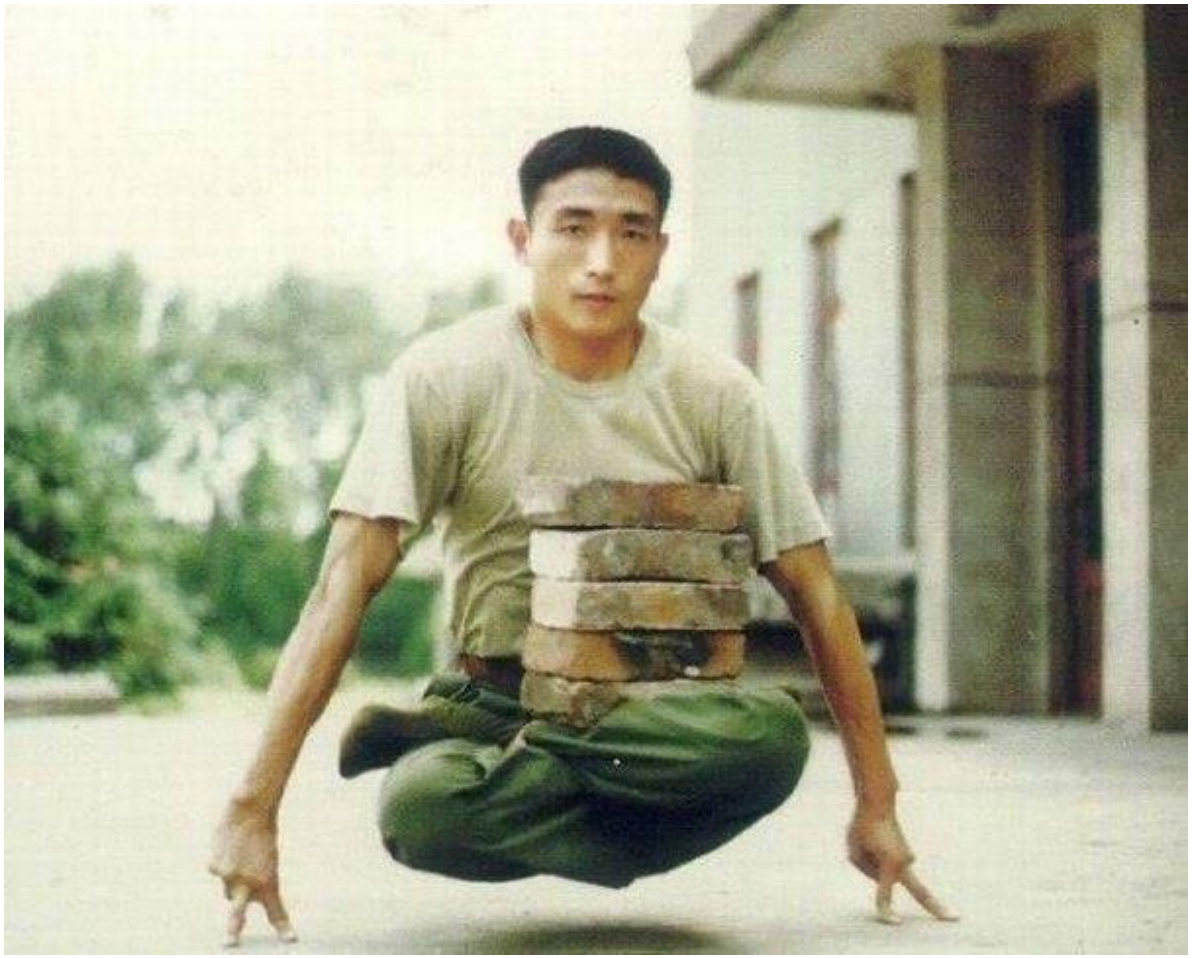
Good coding practice in real life

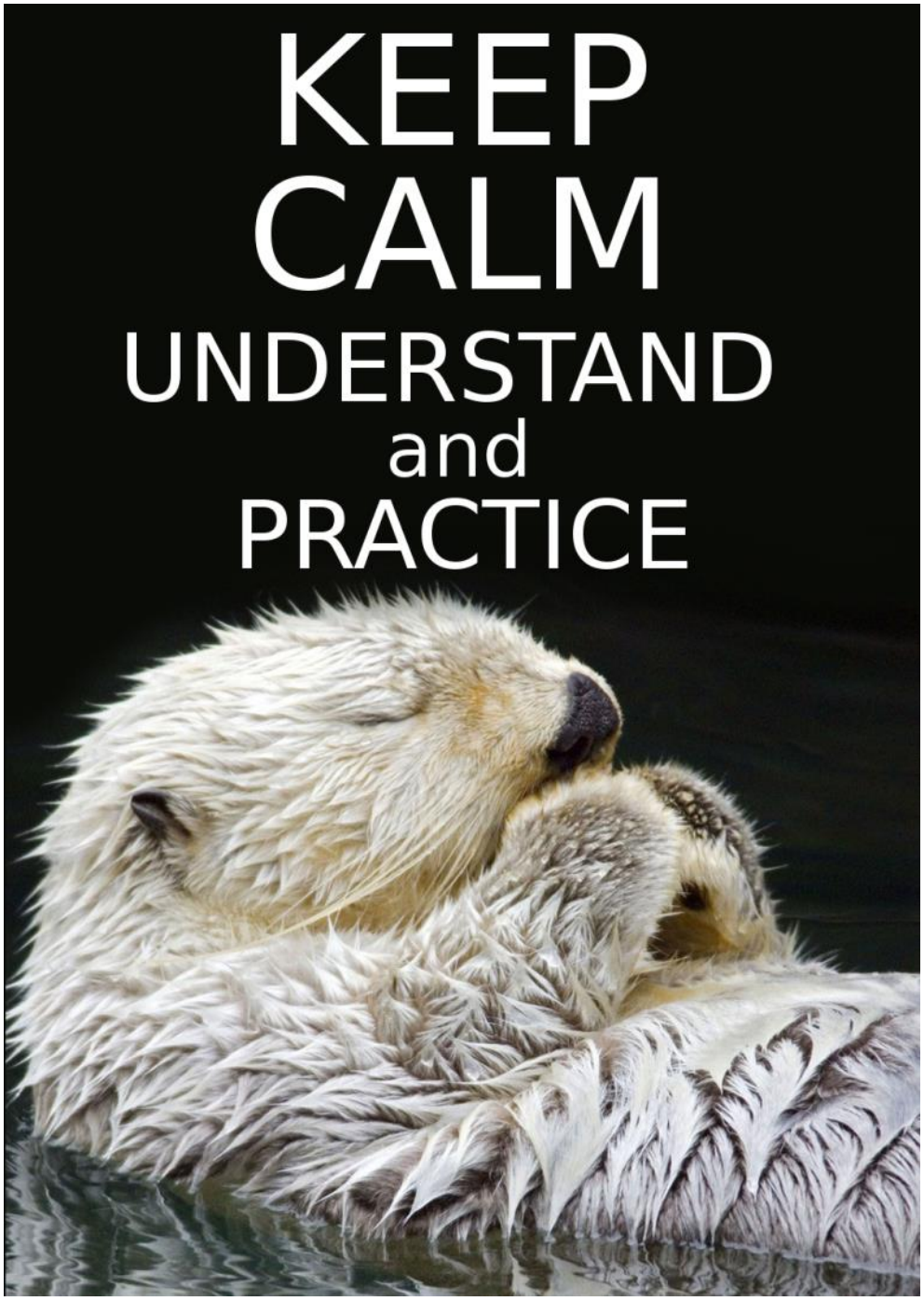
(with a focus on LSP)

Vladimir Alekseichenko

Good practices make life easier

Good practices are not easy




KEEP
CALM
UNDERSTAND
and
PRACTICE

Agenda

Liskov substitution principle

- LSP and OCP.
- The formal definition LSP.
- The informal definition LSP.
- Some examples with violation LSP.
- Signs of violations of the LSP.
- Alternatives to inheritance.

LSP and OCP

Motivation for LSP comes from OCP (at least partly)

– Abstraction and polymorphism are used in standard OO-designs to achieve OCP, but how to use them?

– LSP restricts the use of inheritance, in a way that OCP holds (in the basic OCP design).

– LSP addresses the questions of

- what are the inheritance hierarchies that give designs conforming to OCP?
- what are the common mistakes we make with inheritance regarding OCP?

http://staff.cs.utu.fi/kurssit/Programming-III/LSP(6).pdf

Barbara Liskov

http://news.brown.edu/node/10730

Liskov substitution principle

If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T.

Barbara Liskov, "Data Abstraction and Hierarchy," *SIGPLAN Notices,* 23,5 (May, 1988).

Corollary

Liskov substitution principle

Functions that use pointers or refernces to base classes must be able to use objects of both existing and future derived classes without knowing it.

http://staff.cs.utu.fi/kurssit/Programming-III/LSP(6).pdf

In other words

Liskov substitution principle

Inheritance must be used in a way that any property proved about **supertype** objects also **holds** for **subtype** objects.

http://staff.cs.utu.fi/kurssit/Programming-III/LSP(6).pdf

Liskov substitution principle

Subtypes must be substitutable for their base types.

Liskov substitution principle

If you like the language of contracts, then you might prefer this formulation:

Subtypes must respect the contracts of their supertypes.

http://blog.thecodewhisperer.com/2013/01/08/liskov-substitution-principle-demystified

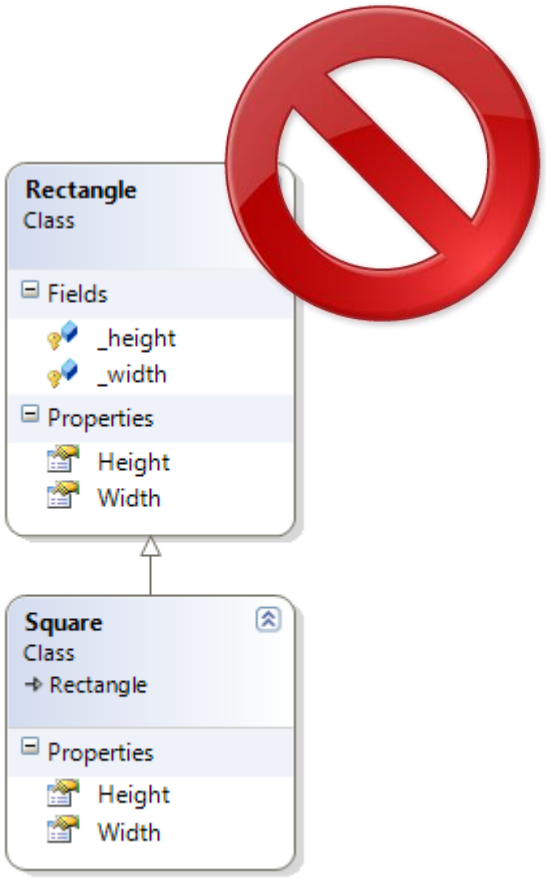
Hmmm… some examples?

The first example

The first example

```
public class Rectangle
{
   protected double _height;
   public virtual double Height
   {
      get { return _height; }
      set { _height = value; }
   }

   protected double _width;
   public virtual double Width
   {
      get { return _width; }
      set { _width = value; }
   }
}
```

```
public class Square : Rectangle
{
   public override double Height
   {
      get { return _height; }
      set {
         _height = value;
         _width = value;
      }
   }

   public override double Width
   {
      get { return _width; }
      set
      {
         _width = value;
         _height = value;
      }
   }
}
```



http://www.objectmentor.com/resources/articles/lsp.pdf

What went wrong?

http://siriusbuzz.com/what-happens-with-that-sirius-lifetime-subscription.php/big-question-mark

What went wrong?

- The **behavior** of a **Square** object **is not** consistent with the **behavior** of a **Rectangle** object.
- Behaviorally, a **Square is not a Rectangle**!
- And it is behavior that software is really all about.

http://www.objectmentor.com/resources/articles/lsp.pdf

How to do well?

http://siriusbuzz.com/what-happens-with-that-sirius-lifetime-subscription.php/big-question-mark

Design by contract

There is a strong relationship between the LSP and the concept of **Design by Contract.**

```
Contract.Requires(_width == value && _height == oldHeight);
```

or

```
if (_width == value && _height == oldHeight)
   throw new Exception("...");
Contract.EndContractBlock();
```

or

```
Debug.Assert(_width == value && _height == oldHeight);
```

The second example

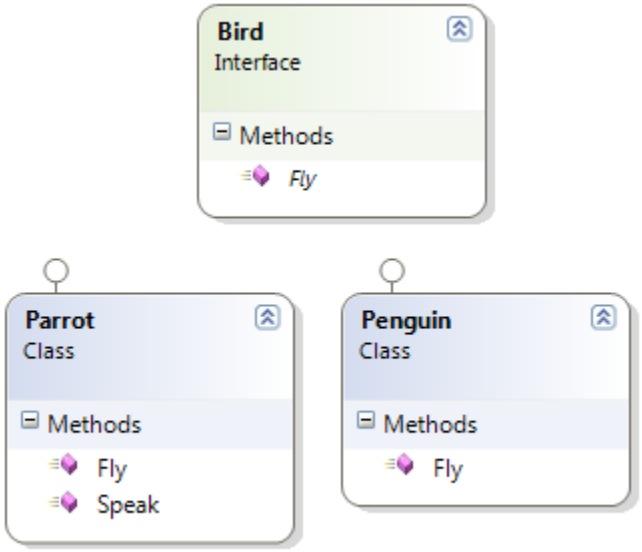
The second example

```
public interface Bird
{
   void Fly();// Bird can fly
}

public class Parrot : Bird // Parrot is a bird
{
   public void Fly()
   {
      //Can fly...
   }

   public void Speak()
   {
      //Can repeat words...
   }
}

public class Penguin : Bird
{
   public void Fly()
   {
      throw new Exception("Penguins don't fly!");
   }
}
```

http://staff.cs.utu.fi/kurssit/Programming-III/LSP(6).pdf

The second example

There are limits to inheritance



http://staff.cs.utu.fi/kurssit/Programming-III/LSP(6).pdf

The second example

```
var mypet = new Parrot();
mypet.Speak();// my pet being a parrot can Speak()
mypet.Fly();// my pet “is-a” bird, can fly
```

```
public void PlayWithBird(Bird abird)
{
   // OK if Parrot.
   //run time error if abird is a Penguin...OOOPS!!
   abird.Fly();
}
```

http://staff.cs.utu.fi/kurssit/Programming-III/LSP(6).pdf

How to do well?

http://siriusbuzz.com/what-happens-with-that-sirius-lifetime-subscription.php/big-question-mark

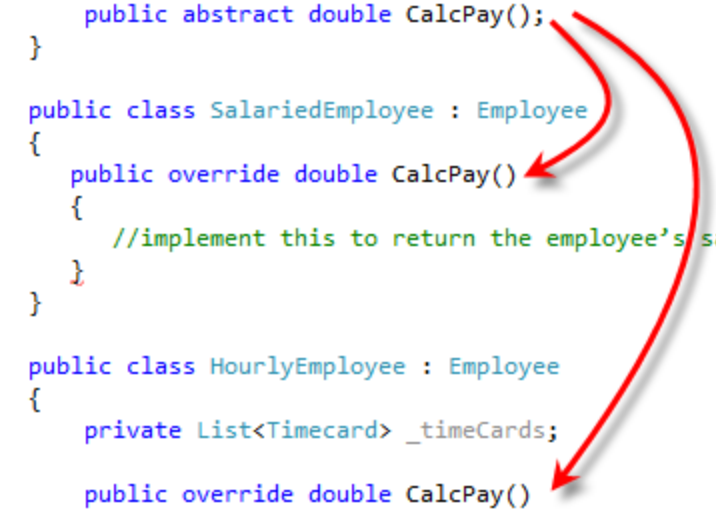
The solution to this problem

```
public interface Bird
{
    void Eat();//Bird can eat
}


public interface FlightBird : Bird
{
    void Fly();//This Bird can fly
}


public class Parrot : FlightBird // Parrot is a bird
{
    public void Fly()
    {
        //Can fly...
    }

    public void Eat()
    {
        //Can eat...
    }

    public void Speak()
    {
        //Can repeat words...
    }
}

public class Penguin : Bird
{
    public void Eat()
    {
        //Can eat...
    }
}
```

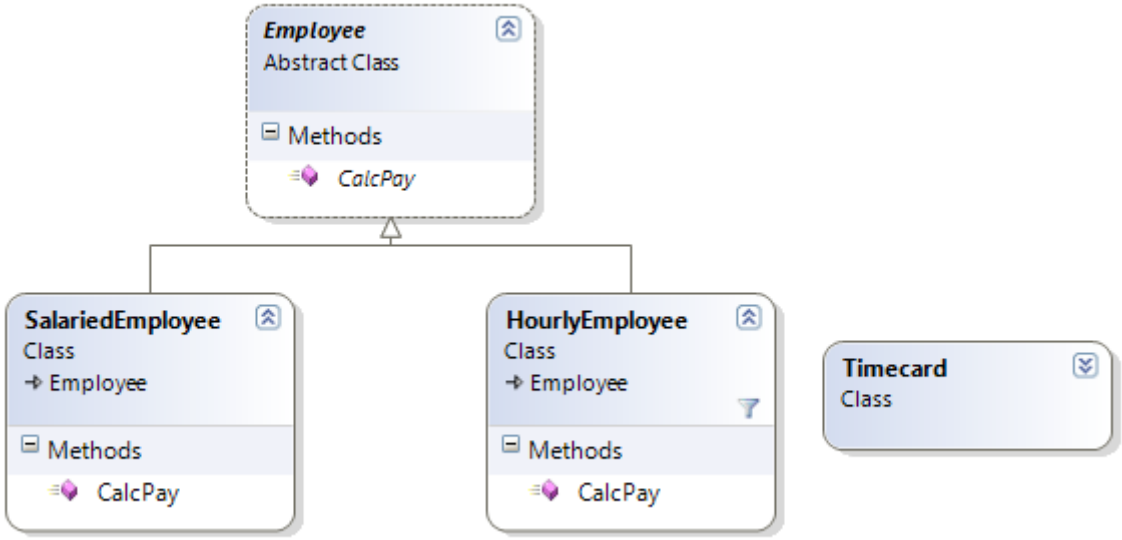
The third example

The third example

```
public abstract class Employee
{
    public abstract double CalcPay();
}

public class SalariedEmployee : Employee
{
   public override double CalcPay()
   {
      //implement this to return the employee's salary
   }
}

public class HourlyEmployee : Employee
{
    private List<Timecard> _timeCards;

    public override double CalcPay()
    {
       //implement it to return the hourly rate times
       //the sum of the hours on this week's time cards.
    }
}

public class Timecard { }
```



The third example



UML for Java (TM) Programmers by *Robert C. Martin*

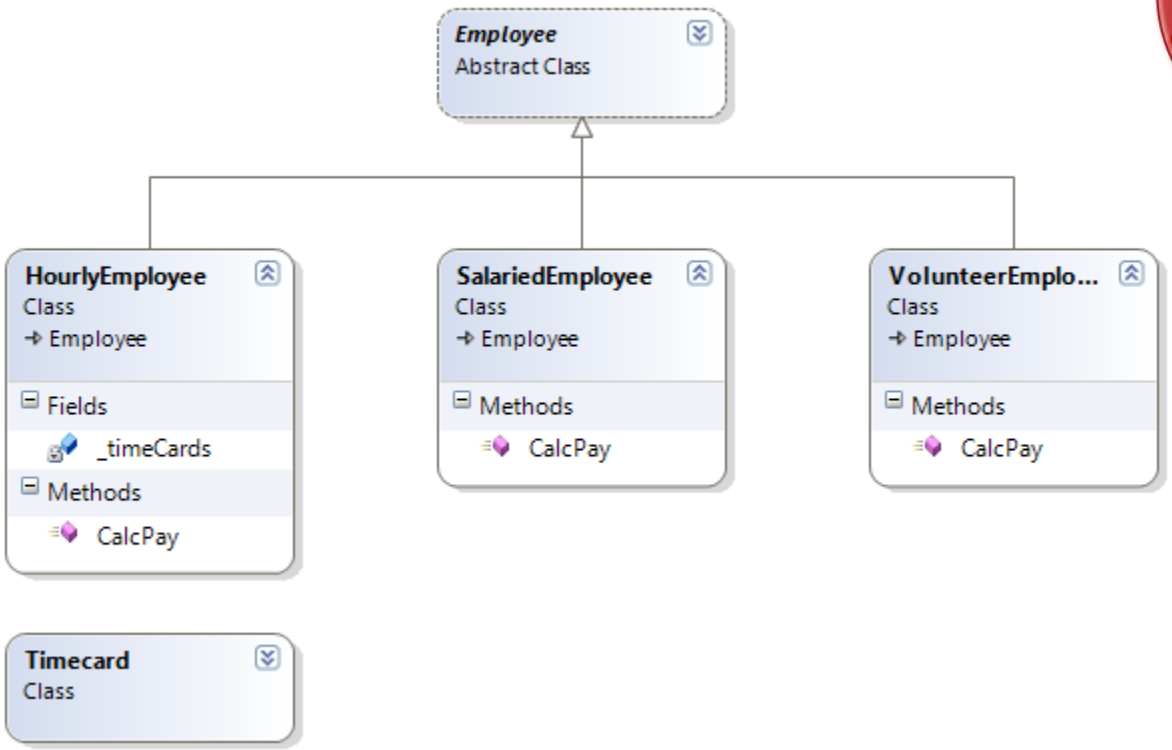
VolunteerEmployee: first step

- What would happen if we decided to add a **VolunteerEmployee**?
- How would we implement **CalcPay**?
- At first this may seem obvious. We'd implement **CalcPay** to return zero as shown below.

```
public class VolunteerEmployee : Employee
{
   public override double CalcPay()
   {
      return 0;
   }
}
```

UML for Java (TM) Programmers by *Robert C. Martin*

VolunteerEmployee: first step



UML for Java (TM) Programmers by *Robert C. Martin*

But is this right?

- Does it make any sense to even call **CalcPay** on a **VolunteerEmployee**?
- After all, by returning zero we are implying that **CalcPay** is a reasonable function to call, and payment is possible.
- We might find ourselves in the embarrasing situation of printing and mailing a paycheck with a gross pay of zero, or some other similar non-sequitur.

VolunteerEmployee: second step

So maybe the best thing to do is to throw an exception, indicating that this function really shouldn't have been called.

```
public class VolunteerEmployee : Employee
{
   public override double CalcPay()
   {
      throw new UnpayableEmployeeException();
   }
}
```

UML for Java (TM) Programmers by *Robert C. Martin*

VolunteerEmployee: second step

To make matters worse, the following code is now illegal.

```
foreach (var employee in employees)
{
   totalPay += employee.CalcPay();
}
```

or

```
double totalPay = employees.Sum(employee => employee.CalcPay());
```

This is ugly, complicated, and distracting.

UML for Java (TM) Programmers by *Robert C. Martin*

VolunteerEmployee: second step

To make it legal we have to wrap the call to **CalcPay** in a *try/catch* block.

```
foreach (var employee in employees)
{
   try
   {
      totalPay += employee.CalcPay();
   }
   catch (UnpayableEmployeeException)
   {
   }
}
return totalPay;
```

VolunteerEmployee: second step

We might easily be tempted to change it to:

```
foreach (var employee in employees)
{
   if (employee is VolunteerEmployee) continue;
   totalPay += employee.CalcPay();
}
return totalPay;
```

UML for Java (TM) Programmers by *Robert C. Martin*

How to do well?

http://siriusbuzz.com/what-happens-with-that-sirius-lifetime-subscription.php/big-question-mark

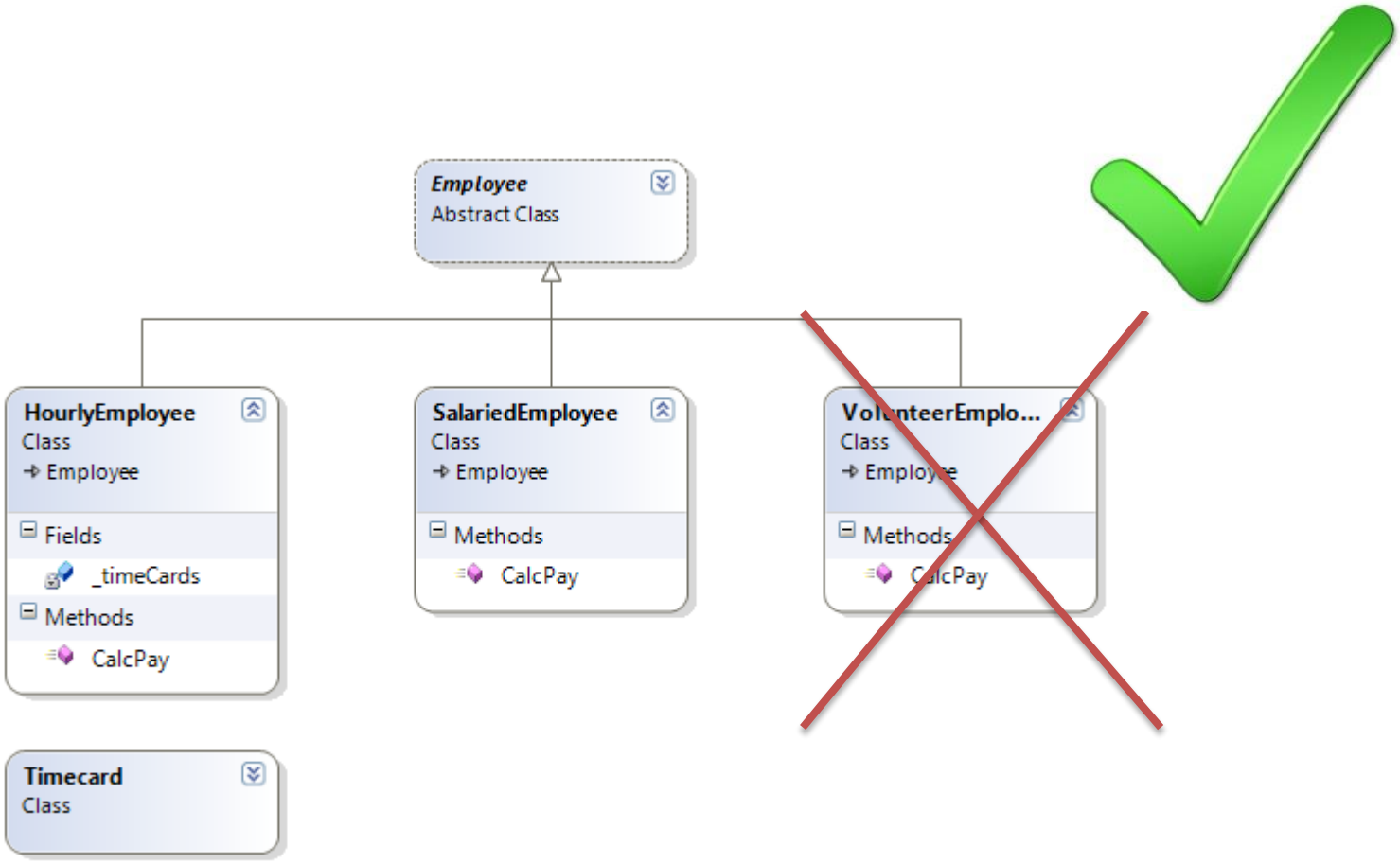
What's the solution to the this problem?

Volunteers are not employees.

It makes no sense to call **CalcPay** on them, so they should not derive from **Employee**, and they should not be passed to functions that need to call **CalcPay**.

UML for Java (TM) Programmers by *Robert C. Martin*

The solution to the this problem

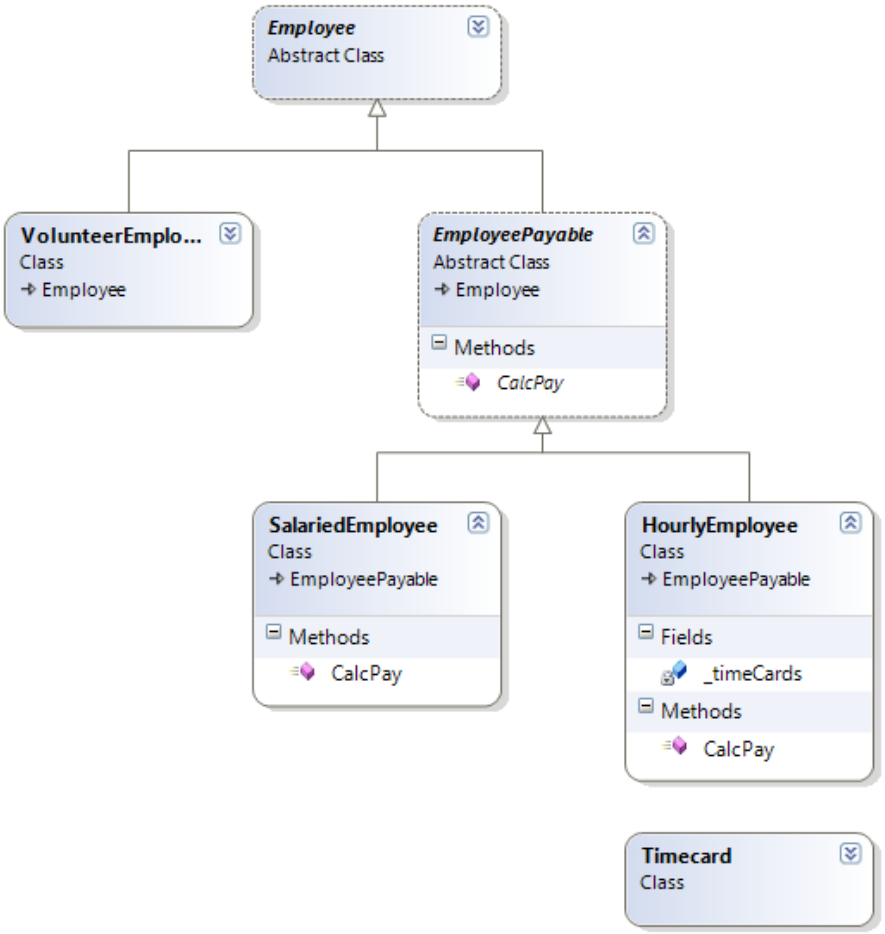


UML for Java (TM) Programmers by *Robert C. Martin*

The solution to the this problem

```
public abstract class Employee { }

public abstract class EmployeePayable : Employee
{
   public abstract double CalcPay();
}

public class SalariedEmployee : EmployeePayable
{
   public override double CalcPay()
   {
      //implement this to return the employee's salary
   }
}

public class HourlyEmployee : EmployeePayable
{
    private List<Timecard> _timeCards;

    public override double CalcPay()
    {
       //implement it to return the hourly rate times
       //the sum of the hours on this week's time cards.
    }
}

public class Timecard { }

public class VolunteerEmployee : Employee {}
```

The solution to the this problem



Signs of violations of the LSP (1)

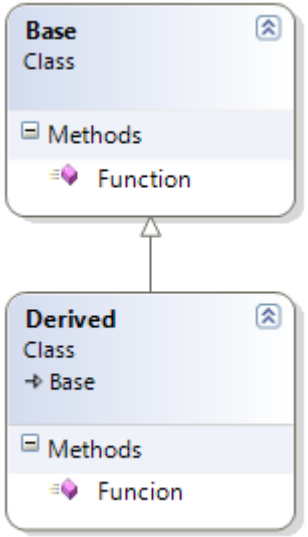
- A subclass doesn't keep all the external observable behavior of its super class.
- A subclass modifies, rather than extends, the external observable behavior of its super class.
- A subclass throws exceptions in an effort to hide certain behavior defined in its super class.

Signs of violations of the LSP (2)

- A subclass that overrides a virtual method defined in its super class using an empty implementation in order to hide certain behavior defined in its super class.
- Method overriding in derived classes is the biggest cause of LSP violations.

Degenerate functions in derivatives

```
public class Base
{
   public void Function()
   {
      //to do something
   }
}

public class Derived : Base
{
   public void Funcion() {}
}
```



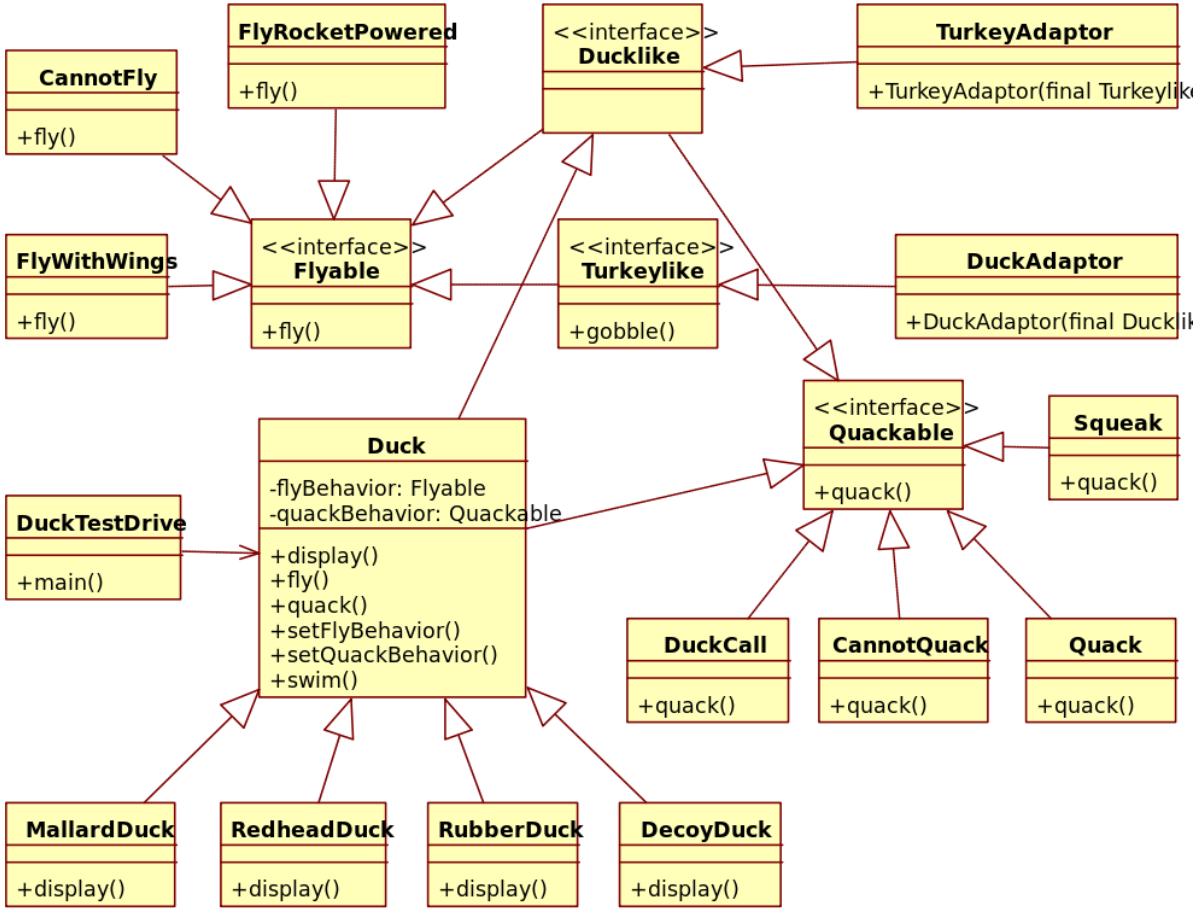
*The presence of degenerate functions in derivatives **is not** alway indicative of an LSP violation, but it's worth looking at them when they occur.*

Agile Software Development, Principles, Patterns, and Practices by *Robert C. Martin*

Alternatives to inheritance

- Delegation
- Composition
- Agregation

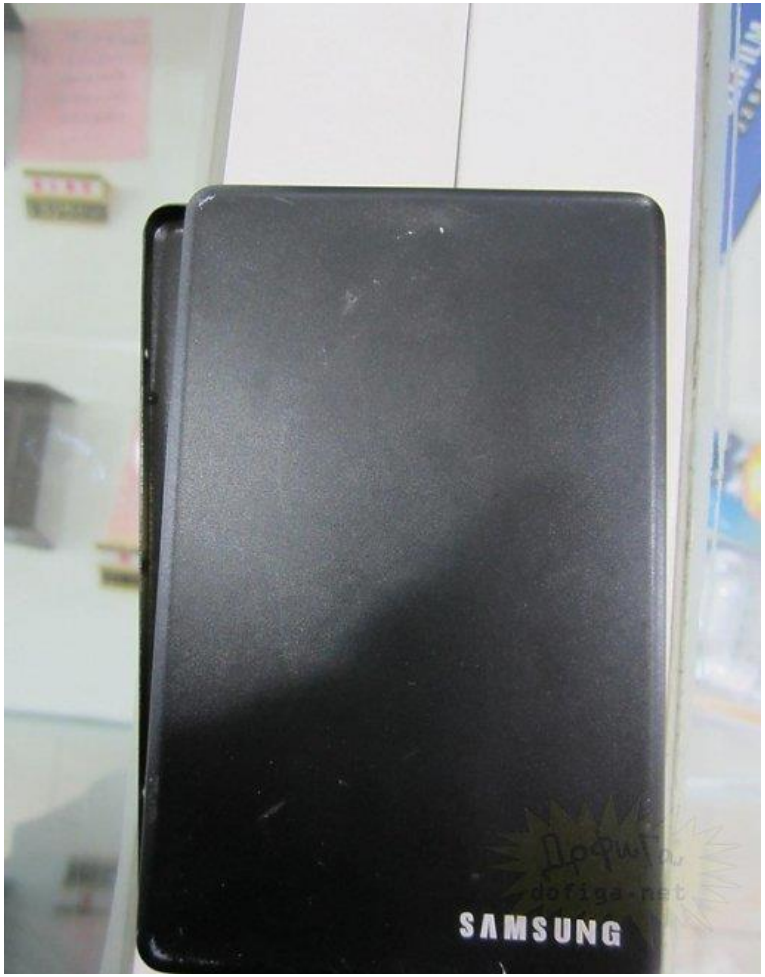
Composition over inheritance



http://en.wikipedia.org/wiki/Composition_over_inheritance

Test your knowledge

Is it a violation of the principle of LSP?



Is it a violation of the principle of LSP?

Is it a violation of the principle of LSP?

```
public class DoubleList<T> : IList<T>
{
   private readonly IList<T> _elements = new List<T>();

   public void Add(T item)
   {
      _elements.Add(item);
      _elements.Add(item);
   }
```

2!

In summary

- Sometimes inheritance just isn't the right thing to do.
- Subclasses should be substitutable for their base classes.
- Subtypes must respect the contracts of their supertypes.

Resources

Books and papers

- **Data abstraction and hierarchy** by *Barbara Liskov*.
- **Object-Oriented Software Construction** by *Bertrand Meyer*.
- **The Liskov Substitution Principle** by *Robert C. Martin.*
- **Design Principles and Patterns** by *Robert C. Martin.*
- **UML for Java (TM) Programmers** by *Robert C. Martin.*
- **Agile Software Development, Principles, Patterns, and Practices** by *Robert C. Martin.*