

SOLID

Principles for Maintainability of Code

Peter Levinsky – IT Roskilde

14.03.2021

Quality Factors

- ISO/IEC 25010

- Functional Suitability
- Performance efficiency
- Compatibility
- Usability
- Reliability
- Security
- Maintainability
- Portability

Quality Factors - Maintainability

- Modularity
- Reusability
- Analysability
- Modifiability
- Testability

S O L I D

- **S** Single Responsibility
- **O** Open / Closed
- **L** Liskov Substitution
- **I** Interface Segregation
- **D** Dependency Injection/Inversion

The Single Responsibility Principle

- Heard of High Cohesion ?
 - one of the GRASP (General Responsibility Assignment Software Patterns) patterns
- Code that changes for the same reasons should be **grouped together**
- Code that changes for different reasons should be **separated**
- Classes should only have **one** main responsibility
- => classes should only have **one** reason to change
- Keep classes small, focused and abstract

The Single Responsibility Principle

Animal

ctor(IWorld iw)

Act()

FoodAround(...)

Sleep(...);

Tanks to Per Laursen

The Single Responsibility Principle

Animal

ctor(IWorld iw)

Act()

FoodAround(...)

Sleep(...);

Models general
animal behavior

Library-like
methods

Tanks to Per Laursen

The Single Responsibility Principle

```
AnimalBehavior  
ctor(IAnimalLibrary  
ial)  
Act()
```

```
AnimalLibrary  
ctor(IWorld iw)  
FoodAround(...)  
Sleep(...);
```


The Open/Closed Principle

- Software entities should be **open** for extension, but **closed** for modification.
- **Open for extension:** behaviour can be extended with new behaviours
- **Closed for modification:** Extension does not require change in the source code for the entity
- “closed for modification that requires clients of the code to change”

The Open/Closed Principle

```
public class Client
{
    public CalculatorV10 _calculator;           // ← quite close for extensions
    public Client()
    {
        _calculator = new CalculatorV10();
    }
    public void ProcessData(Data d)
    {
        int result = _calculator.Calculate(d);
        //...
    }
}
```

The Open/Closed Principle

```
public class Client
{
    public ICalculator _calculator;
    public Client(ICalculator calculator) // ← Now open for extensions
    {
        _calculator = calculator;
    }
    public void ProcessData(Data d)
    {
        int result = _calculator.Calculate(d);
    }
}
```

The Interface Segregation principle

- Like Single Responsibility for classes
 - Interfaces must be small and focused

- An example for CRUD

```
public interface ICreateReadUpdateDelete<T>
{
    void Create(int key, T obj);
    T Read(int key);
    void Update(int key, T obj);
    void Delete(int key);
}
```

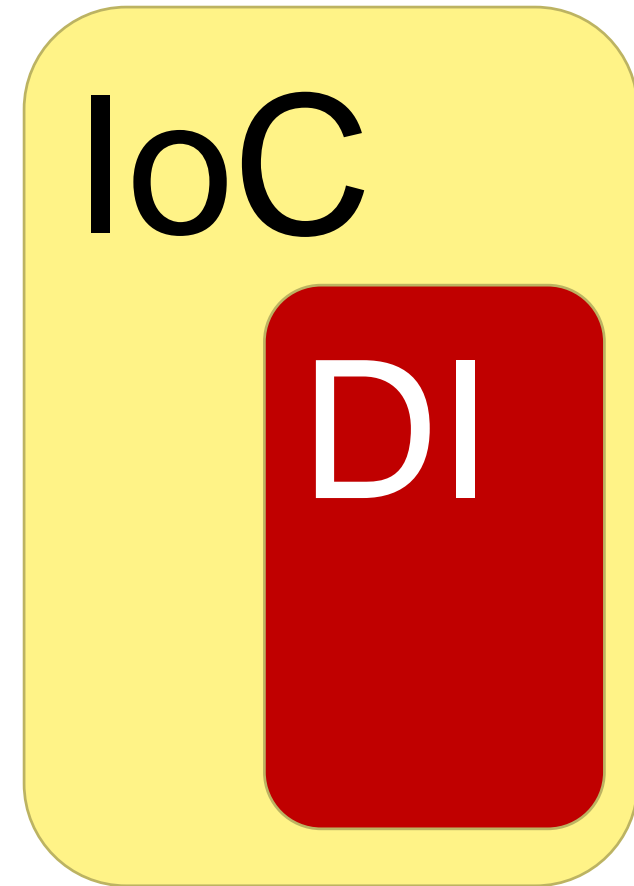
The Interface Segregation principle

- All implementation gets full access to all functionality ! – what if only need get?
- Solution split interface into smaller more focused interfaces e.g.

```
public interface ICreate<T>
{
    void Create(int key, T obj);
}
public interface IRead<T>
{
    T Read(int key);
}
public interface IUpdate<in T>
{
    void Update(int key, T obj);
}
public interface IDelete<T>
{
    void Delete(int key);
}
```

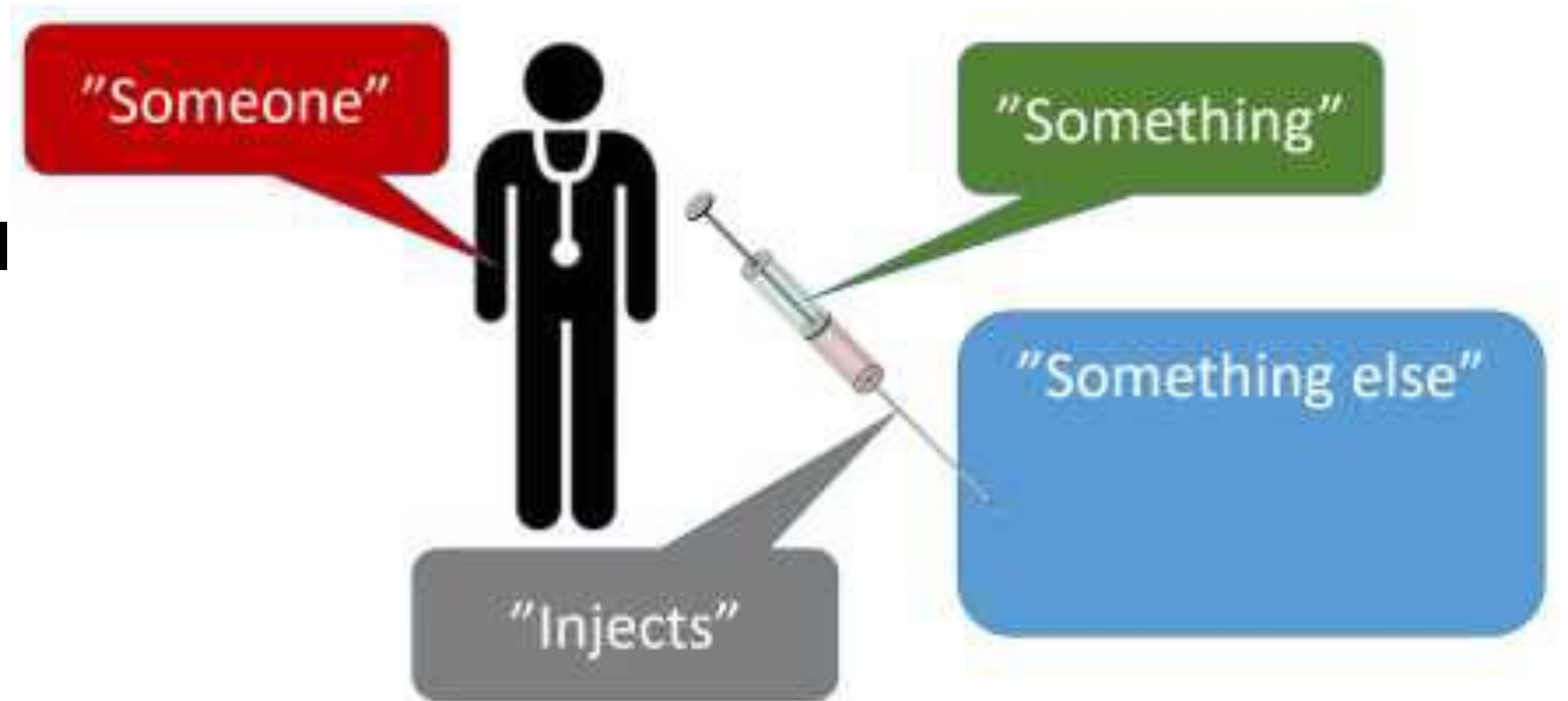
The Dependency Injection/Inversion Principle

- In some SOLID presentations: **Dependency Injection (DI)**
- In other SOLID presentations: **Dependency Inversion**
- **Dependency Inversion**
aka **Inversion of Control (IoC)**



Tanks to Per Laursen

Inversion of Control (IoC)

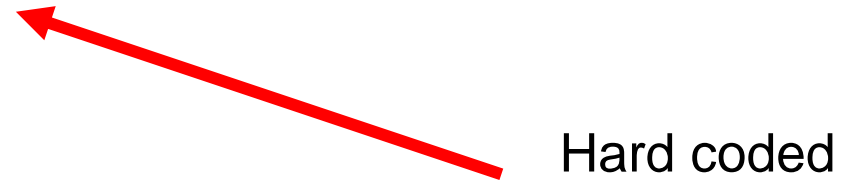


- **Parameter Level**
- **Method Level**
- **Object Level**

Tanks to Per Laursen

Method-level IoC

```
public class Cat : Animal
{
    public override void Act()
    {
        if (FoodAround("Mouse"))
        {
            HuntMice();
        }
        else
        {
            Sleep();
        }
    }
    private void HuntMice() { }
    private void Sleep() { }
}
```



Method-level IoC – cont.

- Template approach

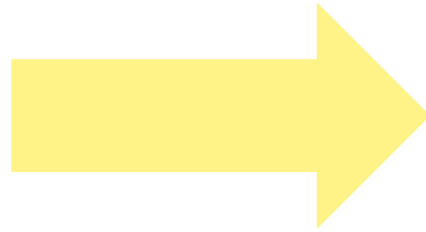
```
public abstract class Animal : IAnimal
{
    public void Act()
    {
        if (FoodAround(PreferredFood()))
        {
            GetFood();
        }
        else
        {
            Idle();
        }
    }
    private bool FoodAround(string food) { return ...; }
    protected abstract string PreferredFood();
    protected abstract void GetFood();
    protected abstract void Idle();
}
```

Method-level IoC – cont.

- *Dependency Injection – parameter level*

```
public int SquareOf4()  
{  
    return 4 * 4;  
}
```

```
public int SquareOf6()  
{  
    return 6 * 6;  
}
```

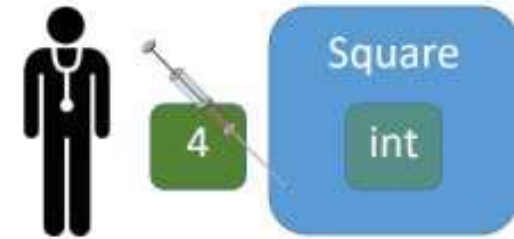


```
public int Square(int n)  
{  
    return n * n;  
}
```

Before



After



Method-level IoC – cont.

```
public class Die
{
    private int _faceValue;
    private static Random _random = new Random();
    public Die()
    {
        Roll();
    }
    public int FaceValue
    {
        get { return _faceValue; }
    }
    public void Roll()
    {
        _faceValue = _random.Next(0,6) + 1;
    }
}
```



```
public class Die
{
    private int _faceValue;
    private int _noOfSides;
    private static Random _random = new Random();
    public Die(int noOfSides)
    {
        _noOfSides = noOfSides;
        Roll();
    }
    public int FaceValue
    {
        get { return _faceValue; }
    }
    public void Roll()
    {
        _faceValue = _random.Next(0, _noOfSides) + 1;
    }
}
```

Method-level IoC – cont.

- *Dependency Injection – method level*

```
public List<int> FilterValues(List<int> values)
{
    List<int> filteredValues = new List<int>();
    foreach (var value in values)
    {
        if (value > 10)
        {
            filteredValues.Add(value);
        }
    }
    return filteredValues;
}
```



```
public List<int> FilterValues(List<int> values, Func<int,bool> condition)
{
    List<int> filteredValues = new List<int>();
    foreach (var value in values)
    {
        if (condition(value))
        {
            filteredValues.Add(value);
        }
    }
    return filteredValues;
}
```

Object-level IoC

- *Dependency Injection – object level*

```
public abstract class Animal : IAnimal
{
    private IWorld TheWorld { get; }
    protected Animal(bool manyOrFew)
    {
        if (manyOrFew)
        {
            TheWorld = new WorldManyAnimals();
        }
        else
        {
            TheWorld = new WorldFewAnimals();
        }
    }
}
```



```
public abstract class Animal : IAnimal
{
    private IWorld TheWorld { get; }
    protected Animal(IWorld theWorld)
    {
        TheWorld = theWorld;
    }
}
```

Two red arrows originate from a point above the right code block. One arrow points to the parameter `IWorld theWorld` in the constructor signature, and the other points to the assignment `TheWorld = theWorld;` inside the constructor body.

Exercises

- **SOLID 1**
- **SOLID 2**

The Liskov Substitution Principle

- Definition

If the class S is a subtype of the class T ,
then objects of type T may be replaced with objects of type S ,
without breaking the program

The Liskov Substitution Principle

- Principle relating to how to create inheritance hierarchies
- Ensures that a client can use subclasses of provided classes **without** changing the expected behaviour

The Liskov Substitution Principle

```
public class S : T
{
    public override void SayHello(string name)
    {
        Console.WriteLine($"Hola {name}");
    }
}
```

The Liskov Substitution Principle

```
public class Client
{
    public void DoSomething(T obj)
    {
        obj.SayHello("Alex");
        obj.SayHello("Betty");
    }
}
```

```
Client aClient = new Client();
aClient.DoSomething(new T());
aClient.DoSomething(new S()); // Have I broken something..?
```

The Liskov Substitution Principle

```
// What does this interface do...?  
public interface IT  
{  
    void SayHello(string name);  
}
```

The Liskov Substitution Principle

```
public interface IT
{
    /// <summary>
    /// Contract: Invoking this method should print a
    /// message on the screen.
    /// The message should
    ///     1) Have a polite greeting nature.
    ///     2) Use the name provided in the argument.
    ///     3) Be in English.
    /// No side effect should occur by calling this method.
    /// </summary>
    void SayHello(string name);
}
```

The Liskov Substitution Principle

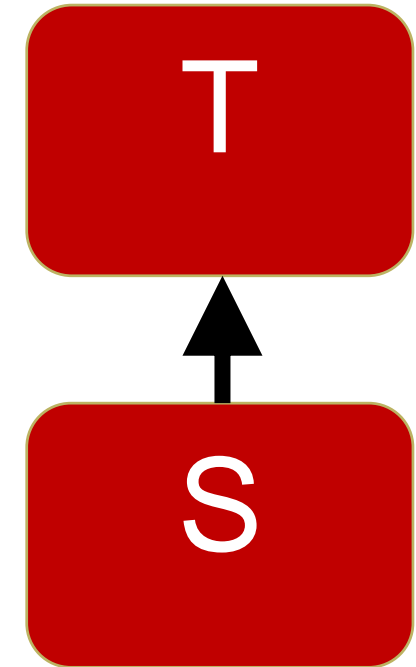
```
public class CheckedGreeting : T
{
    public override void SayHello(string name)
    {
        if (name.Length < 3)
        {
            throw new ArgumentException("Name too short!");
        }

        base.SayHello(name);
    }
}
```



The Liskov Substitution Principle

- **More detailed definition:**
- *If the class **S** is a subtype of the class **T**, then it must always hold that*
 - **Preconditions** in **T** are never strengthened by **S**.
 - **Postconditions** in **T** are never weakened by **S**.
 - **Invariants** in **T** must be preserved by **S**.



The Liskov Substitution Principle

```
// Precondition was strengthened...
public class CheckedGreeting : T
{
    public override void SayHello(string name)
    {
        if (name.Length < 3)
        {
            throw new ArgumentException("Name too short!");
        }

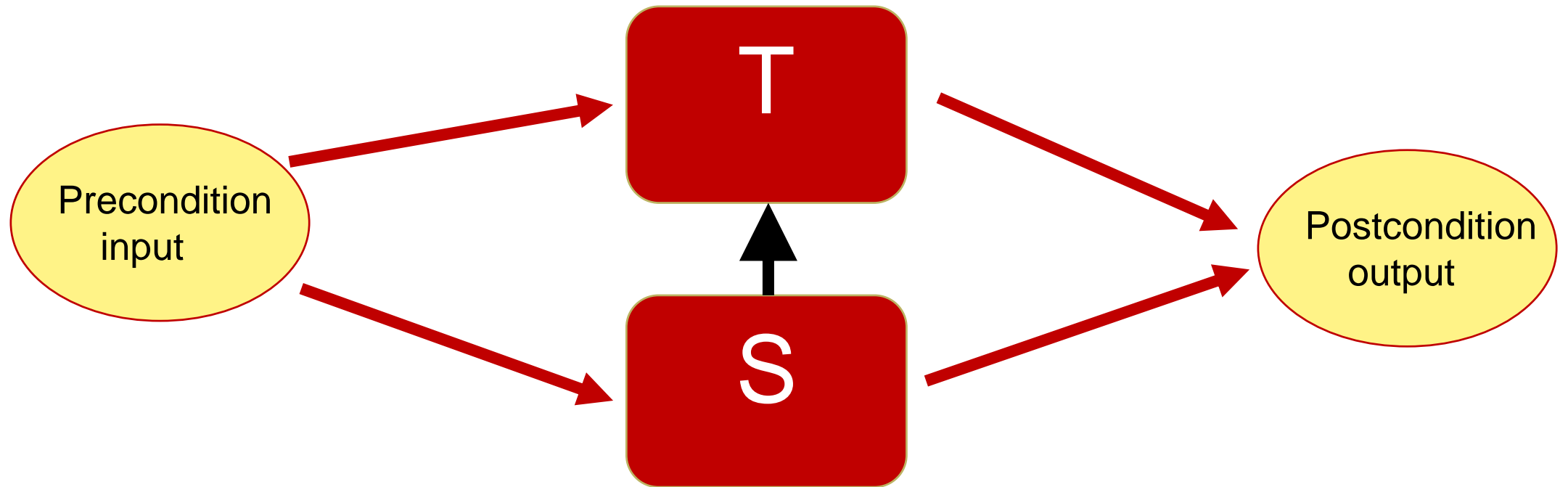
        base.SayHello(name);
    }
}
```

The Liskov Substitution Principle

```
public class Client
{
    public void DoSomething(T obj)
    {
        obj.SayHello("Alex");
        obj.SayHello("Bo");
    }
}
```

```
Client aClient = new Client();
aClient.DoSomething(new T()); // OK
aClient.DoSomething(new CheckedGreeting()); // Oops...
```


The Liskov Substitution Principle



The Liskov Substitution Principle

- Improvement – for precondition

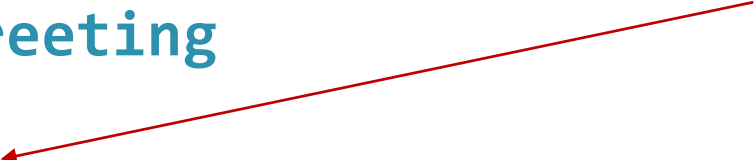
```
public class Name
{
    public string Value { get; }

    public Name(string value)
    {
        if (value.Length < 3) { throw new ...}
        Value = value;
    }
}
```

The Liskov Substitution Principle

```
public class Greeting : IGreeting
{
    public void SayHello(Name name)
    {
        Console.WriteLine($"Hello {name.Value}");
    }
}
```

Input restricted



The Liskov Substitution Principle

- Improvement
 - for postcondition

```
public class Salary
{
    public int Value { get; }

    public Salary(int value)
    {
        if (value < 10000 || 1000000 < value)
            { throw new ...}
        Value = value;
    }
}
```

The Liskov Substitution Principle

```
public interface IEmployee
{
    /// <summary>
    /// Contract: the yearly salary returned must
    /// be a value between 10,000 and 1,000,000
    /// </summary>
    Salary GetYearlySalary();
}
```

output restricted



Exercises

- **SOLID 3**
- **Evt. SOLID 4**

