SOLID

Principles for Maintainability of Code

Peter Levinsky – IT Roskilde

03.10.2024

Quality Factors

ISO/IEC 25010

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

Quality Factors - Maintainability

- Testability
- Reusability
- Analysability
- Modularity
- Modifiability

S O L I D

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

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

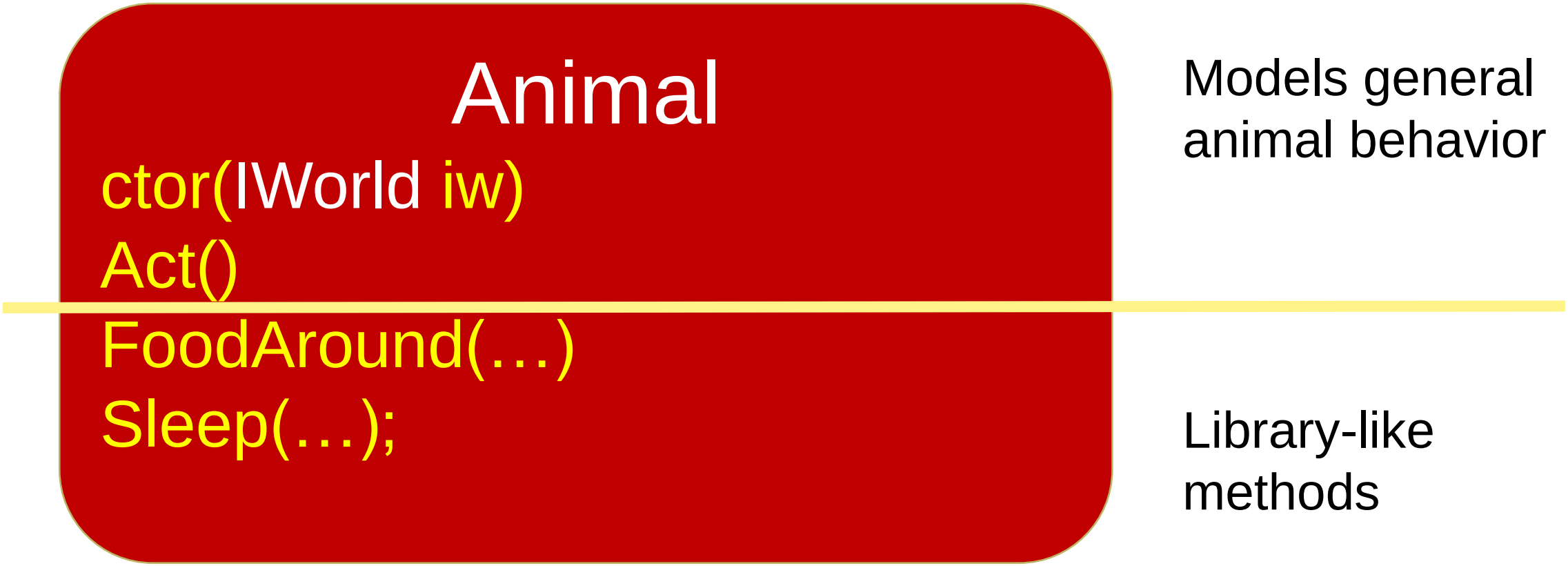
- Code that changes for the same reasons should be **grouped together**
- Code that changes for different reasons should be **separated**

- 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

Tanks to Per Laursen

The Single Responsibility Principle



Tanks to Per Laursen

The Single Responsibility Principle

AnimalBehavior

ctor(IAnimalLibrary ial)

Act()

AnimalLibrary

ctor(IWorld iw)

FoodAround(…)

Sleep(…);

Tanks to Per Laursen

S O L I D

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

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

i.e. “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);
  }
}
```

S O L I D

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

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);
}
```

If you need an interface with all CRUD functions

Example:

```
public interface ICreateReadUpdateDelete<T>:ICreate<T>, IRead<T>, IUpdate<T>, IDelete<T>
{
}
```

S O L I D

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

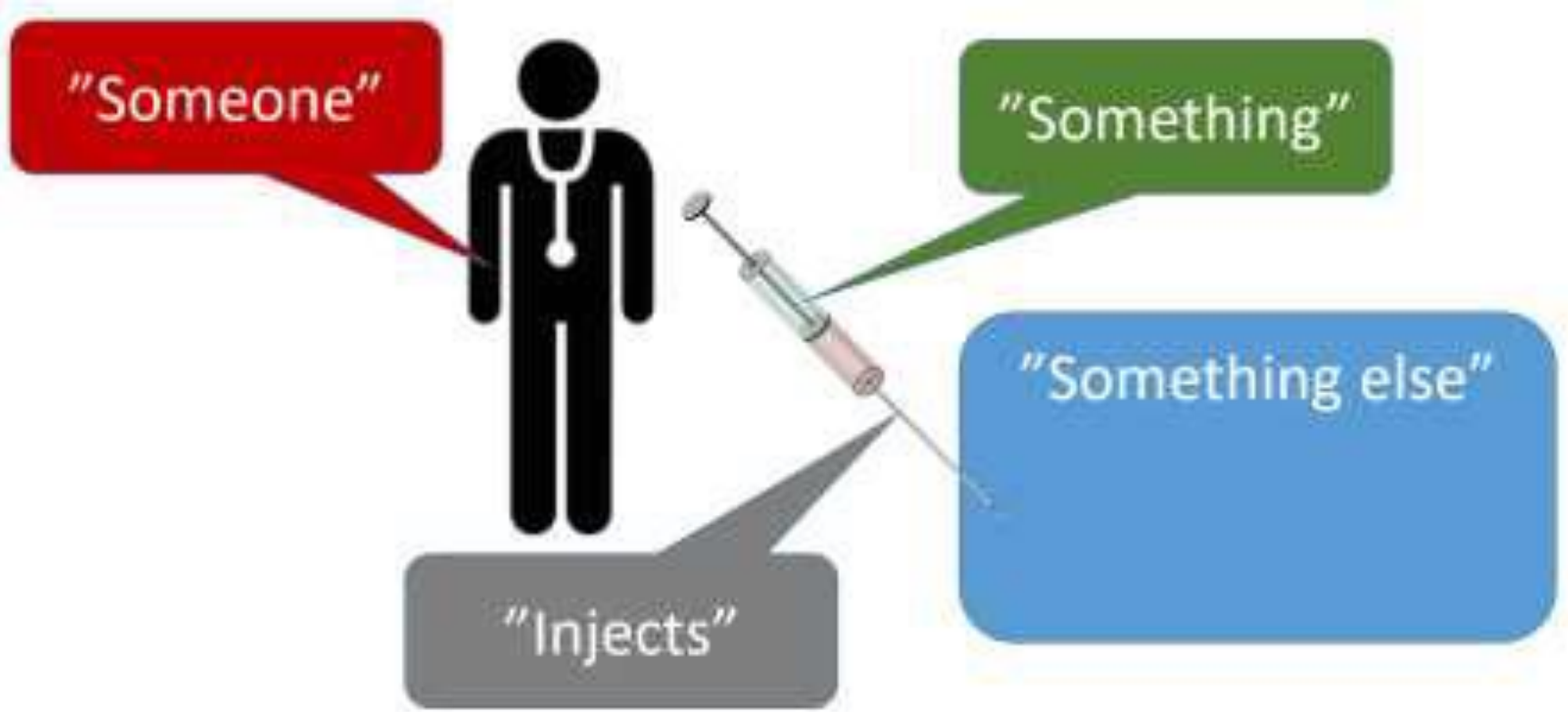
The Dependency Injection/Inversion Principle

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

Tanks to Per Laursen

What is - Inversion of Control (IoC)

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



Tanks to Per Laursen

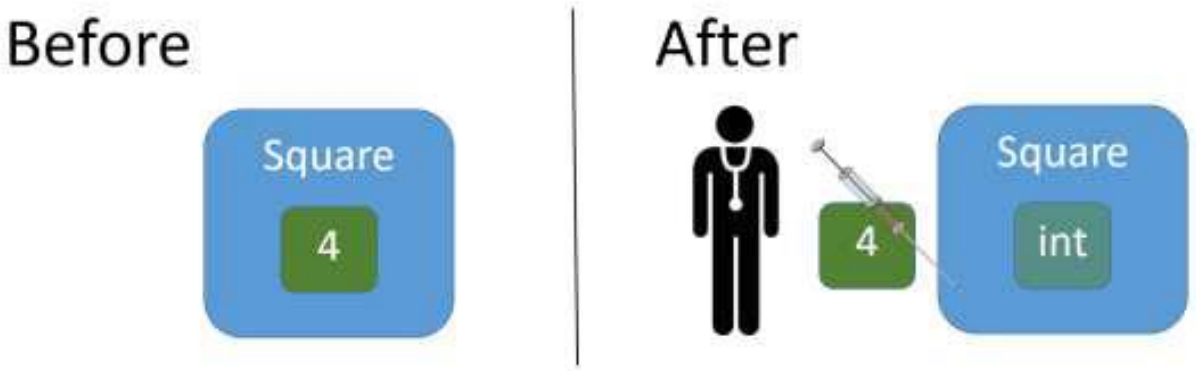
Parameter-level IoC – cont.

- ***Dependency Injection – parameter level***

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

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

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



Parameter-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;
    }
}
```

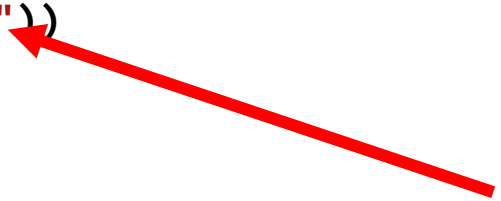


```
public class Die
{
    private int _faceValue;
    private int _noOfSides;
    private static Random _random = new Random();
    public Die(int noOfSides)
    {
        _noOfSides = noOfSides;
        Roll();
     }
     public int FaceValue
     {
         get { return _faceValue; }
      }
   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

Support parameter

Method-level IoC

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



Hard coded

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();
}
```

Use of Template method

Template method to be overridden

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;
}
```

Support method

Object-level IoC

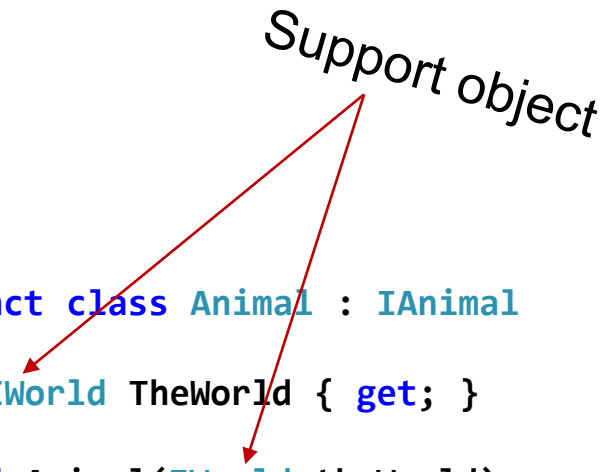
- ***Dependency Injection – object level***

```
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;
     }
}
```



Exercises

- **SOLID 1 + SOLID 2**

og selvfølgelig den obl. opgave