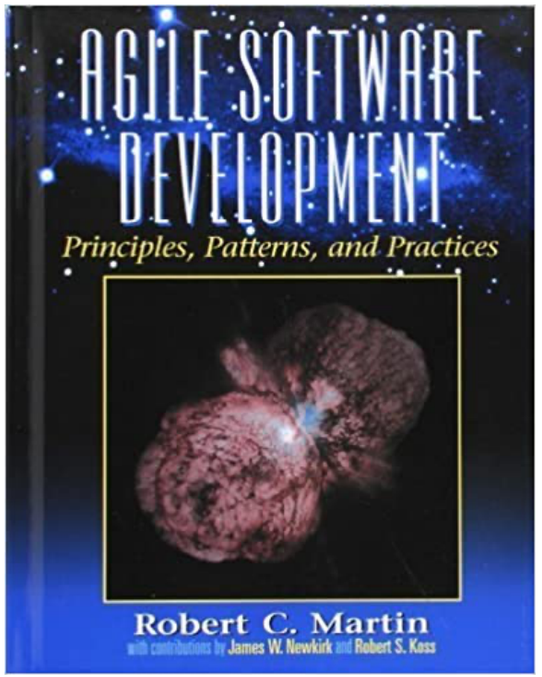
SOLID Prinzipien

Stefan Lieser, CCD Akademie GmbH, 02.02.2022

SOLID Principles

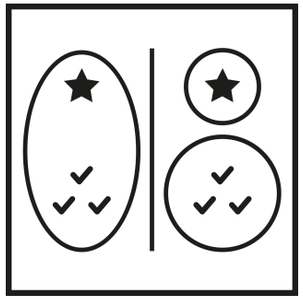
Klingt solide…

- Veröffentlicht 2002 durch **Bob C. Martin** in seinem Buch „Agile Software Development, Principles, Patterns, and Practices"
- Hier noch **nicht** in der Reihenfolge **S**RP/**O**CP/**L**SP/**I**SP/**D**IP
- **Michael Feathers** erkannte die Möglichkeit, das Akronym SOLID zu bilden.



SRP

Single Responsibility Principle



- Eine Funktionseinheit darf **nur einen Grund für Änderungen** liefern.
- Fundamental für den Wert der Wandelbarkeit
- Wenn Aspekte vermischt sind:
  - schwer verständlich
  - erhöhtes Risiko bei Änderungen
  - erschwerte Testbarkeit

SRP - Single Responsibility Principle

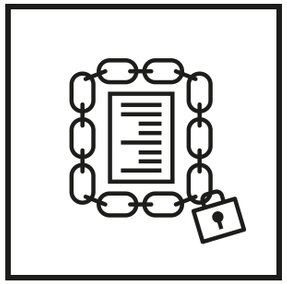
```
public class CsvReader
{
    public IEnumerable<Record> ReadCsvFile(string filename) {
        var lines = File.ReadLines(filename);
        foreach (var line in lines) {
            var values = line.Split(";");
            var record = new Record(values);
            yield return record;
        }
    }
}

public record Record(string[] Values);
```

SRP - Single Responsibility Principle

```
public class CsvReader
{
    public IEnumerable<Record> ReadCsvFile(string filename) {
        var lines = ReadFile(filename);
        var records = CreateRecords(lines);
        return records;
    }

    private static IEnumerable<string> ReadFile(string filename) {
        return File.ReadLines(filename);
    }

    private static IEnumerable<Record> CreateRecords(IEnumerable<string> lines) {
        foreach (var line in lines) {
            var values = line.Split(";");
            var record = new Record(values);
            yield return record;
        }
    }
}
```

OCP

Open Closed Principle



- Eine Funktionseinheit soll **offen sein für Erweiterungen**, aber **abgeschlossen gegenüber Modifikationen**.
- Kann schnell zu generischem Code führen!
  - Keep it simple, stupid!
  - Beware of premature optimization

OCP - Open Closed Principle

```
public IEnumerable<Record> HandleInput(char keyPressed) {
    switch (char.ToLower(keyPressed)) {
        case 'f':
            return FirstPage();
        case 'p':
            return PrevPage();
        case 'n':
            return NextPage();
        case 'l':
            return LastPage();
        default:
            throw new ArgumentException("Unrecognized key", nameof(keyPressed));
    }
}
```

OCP - Open Closed Principle

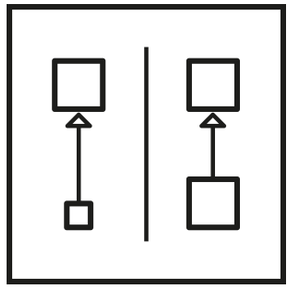
```
public abstract class KeyPressedHandler {
    public abstract bool CanHandleKey(char keyPressed);
    public abstract IEnumerable<Record> GetRecords();
}

public class CsvViewer
{
    private readonly List<KeyPressedHandler> _keyPressedHandlers = new();

    public void AddHandler(KeyPressedHandler keyPressedHandler) {
        _keyPressedHandlers.Add(keyPressedHandler);
    }

    public IEnumerable<Record> HandleInput(char keyPressed) {
        foreach (var keyPressedHandler in _keyPressedHandlers) {
            if (keyPressedHandler.CanHandleKey(keyPressed)) {
                return keyPressedHandler.GetRecords();
            }
        }
        throw new Exception($"No handler found for key '{keyPressed}'");
    }
}
```

OCP - Open Closed Principle

```
public class FirstPageHandler : KeyPressedHandler
{
    public override bool CanHandleKey(char keyPressed) {
        return char.ToLower(keyPressed) == 'f';
    }

    public override IEnumerable<Record> GetRecords() {
        yield break;  // return records for first page
    }
}
```

LSP

Liskov Substitution Principle



- Eine abgeleitete Klasse darf das Verhalten der Basisklasse nicht einschränken.

LSP - Liskov Substitution Principle

```
public class Calculator
{
    public virtual int Magic(int x) {
        return x + 42;
    }
}

public class MagicalCalculator : Calculator
{
    public override int Magic(int x) {
        if (x < 0) {
            throw new Exception("Too much magic needed");
        }
        return base.Magic(x) * 42;
    }
}
```

LSP - Liskov Substitution Principle

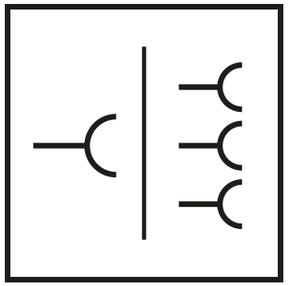
```
public class Magician
{
    private void DoTheTrick(Calculator calculator) {
        var magicResult = calculator.Magic(-42);
        Console.WriteLine(magicResult);
    }

    public void Trick1() {
        DoTheTrick(new Calculator());
    }

    public void Trick2() {
        DoTheTrick(new MagicalCalculator());
    }
}
```

LSP - Liskov Substitution Principle

- Favour Composition over Inheritance!

ISP

Interface Segregation Principle



- Zwei Sichtweisen:
  - Verwender
  - Implementierender
- Verwender eines Interface sollen nicht von Details abhängig gemacht werden, die sie nicht benötigen.

ISP - Interface Segregation Principle

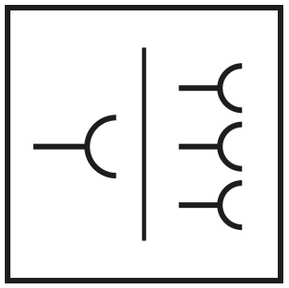
```
public interface IEmail
{
    public void Send(string from, string to, string message);

    public string Receive(string smtpServerName, string username, string password);
}
```

ISP - Interface Segregation Principle

```
public interface IEmailSender
{
    public void Send(string from, string to, string message);
}

public interface IEmailReceiver
{
    public string Receive(string smtpServerName, string username, string password);
}
```

ISP

Interface Segregation Principle



- Implementierende eines Interface sollen nicht zu Details genötigt werden, die sie nicht liefern können.

ISP - Interface Segregation Principle

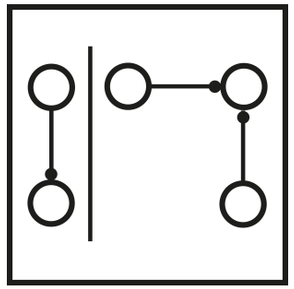
```
public abstract class MessageBase {
    public string From { get; set; }
    public string To { get; set; }
    public string Subject { get; set; }
    public string Message { get; set; }

    public abstract void Send();
}

public class MailMessage : MessageBase {
    public override void Send() {
        // send the message as an email via SMTP
    }
}

public class SmsMessage : MessageBase {
    public override void Send() {
        // send the message as an SMS
        // Subject property is ignored :-(
    }
}
```

ISP - Interface Segregation Principle

```
public abstract class MessageBase {
    public string From { get; set; }
    public string To { get; set; }
    public string Message { get; set; }

    public abstract void Send();
}

public class MailMessage : MessageBase {
    public string Subject { get; set; }

    public override void Send() {
        // send the message as an email via SMTP
    }
}

public class SmsMessage : MessageBase {
    public override void Send() {
        // send the message as an SMS
   }
}
```

DIP

Dependency Inversion Principle

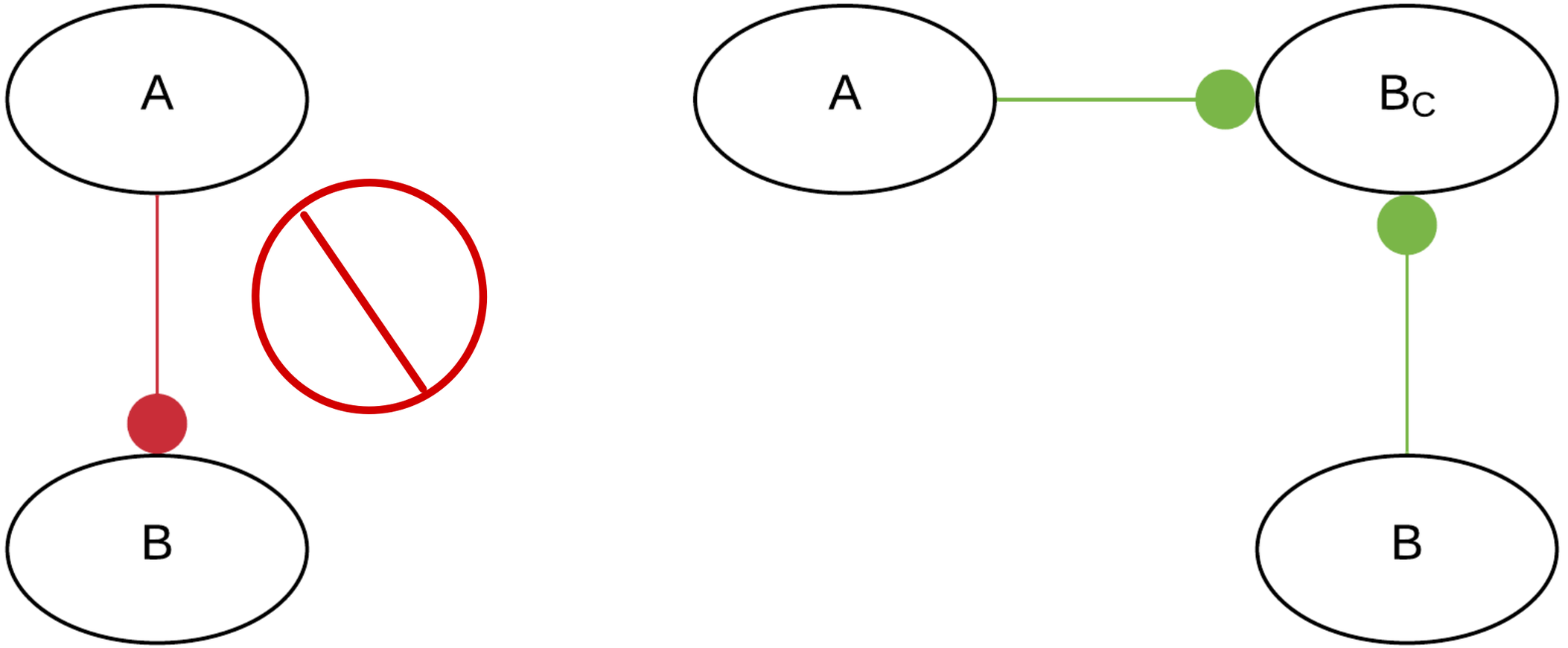


- Eine Klasse soll nicht direkt von einer anderen Klasse abhängig sein, sondern von deren Interface.

- Die Abhängigkeitsrichtung kann umgekehrt werden, durch Einfügen eines Interface.

DIP

Dependency Inversion Principle



DIP - Dependency Inversion Principle

```
public class NewCustomer
{
    public void HandleUseCase(Customer customer) {
        if (customer.Name == "") {
            throw new Exception("Customer name must not be empty");
        }
        customer.Id = Guid.NewGuid().ToString();
        var customerRepository = new CustomerRepository();
        customerRepository.Insert(customer);
    }
}
```

DIP - Dependency Inversion Principle

```
public class NewCustomerDIP
{
    private readonly ICustomerRepository _customerRepository;

    public NewCustomerDIP(ICustomerRepository customerRepository) {
        _customerRepository = customerRepository;
    }

    public void HandleUseCase(Customer customer) {
        if (customer.Name == "") {
            throw new Exception("Customer name must not be empty");
        }
        customer.Id = Guid.NewGuid().ToString();
        _customerRepository.Insert(customer);
    }
}
```

IOSP - Integration Operation Segregation Principle + DIP

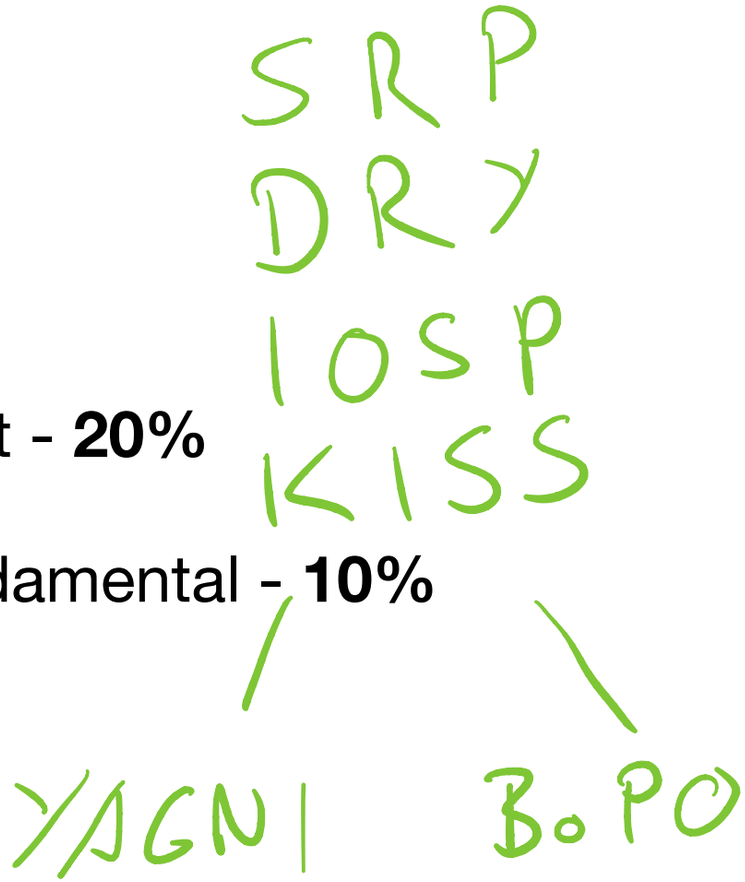
```
public void HandleUseCase(Customer customer) {
    PrepareCustomerForInsert(customer);
    _customerRepository.Insert(customer);
}

private static void PrepareCustomerForInsert(Customer customer) {
    if (customer.Name == "") {
        throw new Exception("Customer name must not be empty");
    }
    customer.Id = Guid.NewGuid().ToString();
}
```

IOSP - Integration Operation Segregation Principle

```
public void HandleUseCase(Customer customer) {
    PrepareCustomerForInsert(customer);
    new CustomerRepository().Insert(customer);
}
```

Relevanz

Sind SOLID die wichtigsten Prinzipien?



- SRP - SEHR wichtig! Grundlage für Wandelbarkeit - **20%**
- OCP - Joa, nicht uninteressant, aber nicht so fundamental - **10%**
- LSP - Nicht relevant in der Praxis - **0%**
- ISP - Nützlich - **10%**
- DIP - Ergänzt durch das IOSP - **10%**
- **Ergibt 50%**
- Jedenfalls keine 100%, somit müssen die SOLID Prinzipien ergänzt werden!

Relevanz

Sind SOLID die wichtigsten Prinzipien?

- SRP - SEHR wichtig! Grundlage für Wandelbarkeit - **20%**
- OCP - Joa, nicht uninteressant, aber nicht so fundamental - **10%**
- LSP - Nicht relevant in der Praxis - **0%**
- ISP - Nützlich - **10%**
- DIP - Ergänzt durch das IOSP - **10%**
- **Ergibt 50%**
- Jedenfalls keine 100%, somit müssen die SOLID Prinzipien ergänzt werden!

Relevanz

Sind SOLID die wichtigsten Prinzipien?

- SRP - SEHR wichtig! Grundlage für Wandelbarkeit - **20%**
- OCP - Joa, nicht uninteressant, aber nicht so fundamental - **10%**
- LSP - Nicht relevant in der Praxis - **0%**
- ISP - Nützlich - **10%**
- DIP - Ergänzt durch das IOSP - **10%**
- **Ergibt 50%**
- Jedenfalls keine 100%, somit müssen die SOLID Prinzipien ergänzt werden!