

# PATRONEN

---

# Patronen?

- “Design patterns represent the best practices used by experienced object-oriented software developers. Design patterns are solutions to general problems that software developers faced during software development. These solutions were obtained by trial and error by numerous software developers over quite a substantial period of time.”  
([http://www.tutorialspoint.com/design\\_pattern/design\\_pattern\\_overview.htm](http://www.tutorialspoint.com/design_pattern/design_pattern_overview.htm))
- Voorbeelden van patronen in C#: <http://www.dofactory.com/net/design-patterns>

# UNIT OF WORK & REPOSITORY PATROON

---

# Repository patroon

- “A Repository mediates between the domain and data mapping layers, acting like an in-memory domain object collection. Client objects construct query specifications declaratively and submit them to Repository for satisfaction. Objects can be added to and removed from the Repository, as they can from a simple collection of objects, and the mapping code encapsulated by the Repository will carry out the appropriate operations behind the scenes. Conceptually, a Repository encapsulates the set of objects persisted in a data store and the operations performed over them, providing a more object-oriented view of the persistence layer. Repository also supports the objective of achieving a clean separation and one-way dependency between the domain and data mapping layers” (<http://www.martinfowler.com/eaCatalog/repository.html> )

# Unit of Work pattern

- “When you're pulling data in and out of a database, it's important to keep track of what you've changed; otherwise, that data won't be written back into the database. Similarly you have to insert new objects you create and remove any objects you delete.
- You can change the database with each change to your object model, but this can lead to lots of very small database calls, which ends up being very slow. Furthermore it requires you to have a transaction open for the whole interaction, which is impractical if you have a business transaction that spans multiple requests. The situation is even worse if you need to keep track of the objects you've read so you can avoid inconsistent reads.
- A Unit of Work keeps track of everything you do during a business transaction that can affect the database. When you're done, it figures out everything that needs to be done to alter the database as a result of your work.”

(<http://www.martinfowler.com/eaCatalog/unitOfWork.html> )

# Codevoorbeeld: interface IEmployeeRepository

```
public interface IEmployeeRepository : IDisposable {  
    IQueryable<employees> All { get; }  
    IQueryable<employees> AllIncluding(params Expression<Func<employees,  
                                     object>>[] includeProperties);  
  
    employees Find(int id);  
    void InsertOrUpdate(employees employee);  
    void Delete(employees employee);  
    void Save();  
}
```

# Codevoorbeeld: EmployeeRepository

```
public class EmployeeRepository : IEmployeeRepository {  
    private HR dbContext;  
    public EmployeeRepository(DbContext dbContext) {  
        this.dbContext = (HR)dbContext;  
    }  
  
    public IQueryable<employees> All {  
        get  
        {  
            return this.dbContext.employees;  
        }  
    }  
}
```

# Codevoorbeeld: EmployeeRepository

```
public IQueryable<employees> AllIncluding(params Expression<Func<employees,
                                         object>>[] includeProperties) {

    IQueryable<employees> query = this.dbContext.employees;
    foreach (var includeProperty in includeProperties) {
        query = query.Include(includeProperty);
    }
    return query;
}

public void Delete(employees employee) {
    this.dbContext.employees.Remove(employee);
}
```



# Codevoorbeeld: EmployeeRepository

```
public void Dispose() {  
    this.dbContext.Dispose();  
}  
  
public employees Find(int id) {  
    return dbContext.employees.Where(x => x.id == id).Select(x =>  
        x).FirstOrDefault();  
}  
  
public void InsertOrUpdate(employees employee) {  
    if (employee.id == default(int)) {  
        this.dbContext.employees.Add(employee);  
    }  
}
```

# Codevoorbeeld: EmployeeRepository

```
        else {  
            this.dbContext.Entry(employee).State = EntityState.Modified;  
        }  
    }  
  
    public void Save() {  
        this.dbContext.SaveChanges();  
    }  
}
```

# Codevoorbeeld: IUnitOfWork

```
public interface IUnitOfWork {  
    IEmployeeRepository EmployeesRepo { get; }  
}
```

# Codevoorbeeld: UnitOfWork

```
public class UnitOfWork : IUnitOfWork {  
    private IEmployeeRepository employeesRepo;  
    private DbContext hr;  
    public UnitOfWork(DbContext hr) {  
        this.hr = hr;  
    }  
    public IEmployeeRepository EmployeesRepo {  
        get {  
            if (this.employeesRepo == null) {  
                this.employeesRepo = new EmployeeRepository(hr);  
            }  
            return employeesRepo;  
        }  
    }  
}
```

# Codevoorbeeld: Program.cs

```
static void Main(string[] args) {  
    UnitOfWork uoW = new UnitOfWork(new HR());  
    employees emp1 = uoW.EmployeesRepo.Find(3);  
    Console.WriteLine(emp1.FirstName);  
    IQueryable<employees> employees = uoW.EmployeesRepo.All;  
    var employeesBornIn1973 = uoW.EmployeesRepo.AllIncluding().Where( x =>  
        x.DayOfBirth.Year.Equals(1973));  
    foreach (var s in employeesBornIn1973) {  
        Console.WriteLine(s.FirstName + " " + s.LastName);  
    }  
}
```

# Codevoorbeeld: Program.cs

```
employees empNew = new employees {  
    FirstName = "Tiesj",  
    LastName = "Benoot",  
    DayOfBirth = new DateTime(1990,3,11)  
};  
uow.EmployeesRepo.InsertOrUpdate(empNew);  
uow.EmployeesRepo.Save();  
Console.ReadKey();  
Console.ReadKey();  
}
```

# Demo

# DEPENDENCY INJECTION

---



# Tightly vs loosely coupled

- Applicatie is tightly coupled als zijn klassen afhankelijk zijn van andere **concrete** klassen

# Voorbeeld van tightly coupled applicatie

```
public class Commerce {  
    private PaymentProcessor _processor;  
    public Commerce() {  
        _processor = new PaymentProcessor();  
    }  
  
    public void ProcessPayment() {  
        // do some work  
        _processor.ProcessPayment();  
    }  
}
```

# Voorbeeld van tightly coupled applicatie

```
public class PaymentProcessor {  
    public void ProcessPayment() {  
        // process payment from customer  
    }  
}
```

# Nadelen

- Commerce moet ervoor zorgen dat de klasse `PaymentProcessor` geïntantieerd wordt
- Code zal niet compileren als niet alle afhankelijkheden in orde gebracht zijn
- En de klasse moet gekend zijn voor het compileren van de code
- Stel dat er meerdere concrete implementaties van `PaymentProcessor` klasse zijn – eentje voor elke soort betaling => dan heb je een probleem met dit stuk code. Je kan niet makkelijk switchen van betaalmethode.

# Afhankelijkheden op verschillende niveaus

- Stel dat klasse PaymentProcessor ook nog een afhankelijkheid heeft naar een CurrencyConverter klasse. Commerce klasse blijft ongewijzigd.

```
public class PaymentProcessor {  
    private CurrencyConverter _cc;  
    public PaymentProcessor() {  
        _cc = new CurrencyConverter();  
    }  
  
    public void ProcessPayment() {  
        double convertedCurrency = _cc.ConvertToLocalCurrency();  
        // do some work  
    }  
}
```

# Afhankelijkheden op verschillende niveaus

```
public class CurrencyConverter {  
    public decimal ConvertToLocalCurrency() {  
        // return converted value  
    }  
}
```

# Probleem met unit testen van de code

- Stel nu dat CurrencyConverter een methode bevat die telkens de methode aanroepen wordt een databank op het internet raadpleegt
- Probleem voor testen, want dit duurt relatief lang om te testen en er kunnen ook onvoorziene fouten optreden door slechte internetconnectie bijv, of omdat de webservice die je raadpleegt plat ligt

# Betere aanpak

- Is om met interfaces te werken ipv concrete klassen
- Je kan interfaces “mocken”, zodat tijdens testen niet telkens een databank of webservice moet aangeroepen worden
- Het is ook een good practice om geen “new” te gebruiken in klassen. Wordt “avoid newing-up objects” genoemd
- Beter is om te werken via een factory of iets anders om dit te bewerkstelligen



# Enter loosely coupled code

- Loosely coupled applicatie heeft in principe een goede afscherming tussen zijn klassen, modules enz...
- Als we iets zouden wijzigen aan een deel van de applicatie moeten we enkel de relevante modules updaten. We moeten geen veranderingen aanbrengen in de ganse code.
- Als we kijken naar dependency injection:
  - Een afhankelijke klasse moet losgekoppeld zijn van zijn afhankelijke klassen zodat de afhankelijke klasse zich geen zorgen moet maken over het instantiëren van de afhankelijkheden
  - De gebruiker van de afhankelijke klasse moet alle afhankelijkheden voorzien wanneer hij/zij het instantieert

# Waarom losgekoppelde code?

## 1. Uitbreidbaarheid

- Door losgekoppelde code kunnen we snel reageren op nieuwe vereisten.
- Een nieuwe feature toevoegen kan je door deze in te pluggen in de applicatie (zonder veel van de bestaande code te moeten herschrijven)

## 2. Unit testing

- Als de code losgekoppeld is, dan kan je in principe elk stuk code unit testen

## 3. Late binding

- Door losgekoppelde code kan men ook dynamisch modules laden wanneer de applicatie wordt uitgevoerd
- Je kan bijv. verschillende componenten shippen naar verschillende klanten, en de code neemt de goede componenten wanneer ze gedeployed wordt.

# Waarom losgekoppelde code?

## 4. Parallele ontwikkeling

- Als modules niet direct van elkaar afhankelijk zijn kunnen ze makkelijker door verschillende leden van het dev team ontwikkeld worden
- En veranderingen aan de ene module zullen geen impact hebben op de andere module

## 5. Onderhoudbaarheid

- Goede code is makkelijk onderhoudbare code => losgekoppelde code helpt ons hierbij!

# Impliciete vs expliciete afhankelijkheden

- Een afhankelijkheid is impliciet als deze alleen bestaat in de code van de klasse en niet in de publieke interface.
- Daardoor => als een klasse geïntantieerd wordt, weten we niet dat er afhankelijkheden bestaan die de klasse nodig heeft om alles te kunnen uitvoeren en daardoor kan de code falen
- Klassen met impliciete afhankelijkheden zijn moeilijker te onderhouden dan expliciete afhankelijkheden en zijn ook moeilijker te testen.

# Codevoorbeeld van impliciete afhankelijkheid

```
class Program {  
    static void Main(string[] args) {  
        decimal paidValue = 453.23m;  
        var commerce = new Commerce();  
        commerce.ProcessCustomerPayment(paidValue);  
    }  
}  
  
public class Commerce {  
    public void ProcessCustomerPayment(decimal paidValue) {  
        var currencyConverter = new CurrencyConverter();  
        var paymentProcessorv = new PaymentProcessor();  
        decimal currencyValue = currencyConverter.ConvertCurrency(paidValue);  
        paymentProcessor.ProcessPayment(currencyValue);  
        // do some work  
    }  
}
```

# Codevoorbeeld van impliciete afhankelijkheid

```
public class PaymentProcessor {  
    public void ProcessPayment(decimal value)  
    {  
        // do some work  
    }  
}  
  
public class CurrencyConverter {  
    public decimal ConvertCurrency(decimal value)  
    {  
        //convert value to local currency and return  
        return value;  
    }  
}
```

# Expliciete afhankelijkheden

- Expliciete afhankelijkheid betekent dat een klasse al zijn klasse-afhankelijkheden toont in zijn constructor.
- Een expliciete afhankelijkheid wordt meestal gedeclareerd als een interface in de constructor. Daardoor kan de afhankelijkheid makkelijk veranderd worden met een andere implementatie van de interface (tijdens productie, testing of debugging).
- Makkelijker om code te onderhouden
- Dit noemt men constructor injection

# Zelfde codevoorbeeld maar met expliciete afhankelijkheden

```
class Program {  
    static void Main(string[] args) {  
        decimal paidValue = 453.23m;  
        var commerce = new Commerce(new PaymentProcessor(), new CurrencyConverter());  
        commerce.ProcessCustomerPayment(paidValue);  
    }  
}  
  
public class Commerce {  
    IPaymentProcessor _paymentProcessor;  
    ICurrencyConverter _currencyConverter;  
  
    public Commerce(IPaymentProcessor paymentProcessor, ICurrencyConverter currencyConverter) {  
        _paymentProcessor = paymentProcessor;  
        _currencyConverter = currencyConverter;  
    }  
}
```



# Zelfde codevoorbeeld maar met expliciete afhankelijkheden

```
public void ProcessCustomerPayment(decimal paidValue) {  
    // do some work  
    decimal currencyValue = _currencyConverter.ConvertCurrency(paidValue);  
    _paymentProcessor.ProcessPayment(currencyValue);  
}  
  
}  
  
//Dependency Classes and their respective Interfaces  
public interface IPaymentProcessor {  
    void ProcessPayment(decimal value);  
}
```

# Zelfde codevoorbeeld maar met expliciete afhankelijkheden

```
public class PaymentProcessor : IPaymentProcessor {  
    public void ProcessPayment(decimal value) {  
        // do some work  
    }  
}  
  
public interface ICurrencyConverter {  
    decimal ConvertCurrency(decimal value);  
}  
  
public class CurrencyConverter : ICurrencyConverter {  
    public decimal ConvertCurrency(decimal value) {  
        //convert value to local currency  
        return value;  
    }  
}
```

# Nog een ander codevoorbeeld

```
public class Program {  
    Order customerOrder = new Order { Id = 1, ProductName = "Product", Quantity = 3, UnitPrice  
= 75 };  
    Commerce commerce = new Commerce(new CreditCardProcessor(new CurrencyConverter()),  
                                     new EmailNotifier(), new TextLogger());  
    commerce.ProcessOrder(customerOrder);  
}
```

```
public class Commerce {  
    private PaymentProcessor _paymentProcessor;  
    private NotificationManager _notificationManager;  
    private Logger _logger;
```

# Nog een ander codevoorbeeld

```
public Commerce(PaymentProcessor paymentProcessor, NotificationManager notificationManager, Logger logger) {  
    _paymentProcessor = paymentProcessor;  
    _notificationManager = notificationManager;  
    _logger = logger;  
}  
  
public void ProcessOrder(Order order) {  
    decimal paidAmount = order.UnitPrice * order.Quantity;  
    bool paymentSuccessfull = _paymentProcessor.ProcessPayment(paidAmount);  
    if(paymentSuccessfull) {  
        _notificationManager.NotifyCustomer(notification: "payment successful");  
    }  
    else {  
        _notificationManager.NotifyCustomer(notification: "payment failed");  
        _logger.Log(errorMessage: "payment failed");  
    }  
}  
}
```

# Mogelijk probleem

- We hebben nog maar een kleine applicatie, maar alle afhankelijkheden van de Commerce klasse worden nu in Program klasse geïnstantieerd.
- Wat gebeurt er als we met veel meer klassen en afhankelijkheden werken? Niet meer te doen om dat manueel te blijven doen.
- Dependency Injection Container to the rescue!

# Dependency Injection Container

- DI Container neemt de verantwoordelijkheid om de afhankelijkheden te instantiëren en te voorzien, onafhankelijk van het aantal afhankelijkheden
- Container moet natuurlijk wel weten wat de afhankelijkheden zijn.
- We moeten container meegeven welke concrete implementatie er moet gereturned worden als we een afhankelijkheid van de vorm `ISomeInterface` injecteren in de code.
- In codevoorbeeld gebruiken we Autofac. Er bestaan verschillende andere DI Containers.

# Codevoorbeeld met DI Container

```
public class IoCBuilder {  
    internal static IContainer Build() {  
        ContainerBuilder builder = new ContainerBuilder();  
        RegisterTypes(builder);  
        return builder.Build();  
    }  
  
    private static void RegisterTypes(ContainerBuilder builder) {  
        builder.RegisterType<Commerce>();  
        builder.RegisterType<CurrencyConverter>().As<ICurrencyConverter>();  
        builder.RegisterType<TextLogger>().As<ILogger>();  
        builder.RegisterType<EmailNotifier>().As<INotificationManager>();  
        builder.RegisterType<CreditCardProcessor>().As<IPaymentProcessor>();  
    }  
}
```

# Codevoorbeeld met DI Container

```
public class Commerce {  
    private IPaymentProcessor _paymentProcessor;  
    private INotificationManager _notificationManager;  
    private ILogger _logger;  
  
    public Commerce(IPaymentProcessor paymentProcessor, INotificationManager notificationManager, ILogger logger) {  
        _paymentProcessor = paymentProcessor;  
        _notificationManager = notificationManager;  
        _logger = logger;  
    }  
  
    public void ProcessOrder(Order order)  
    {  
        decimal paidAmount = order.UnitPrice * order.Quantity;  
        bool paymentSuccessful = _paymentProcessor.ProcessPayment(paidAmount);  
    }  
}
```



# Codevoorbeeld met DI Container

```
        if(paymentSuccessfull) {
            _notificationManager.NotifyCustomer(notification: "payment successful");
        }
        else {
            _notificationManager.NotifyCustomer(notification: "payment failed");
            _logger.Log(errorMessage: "payment failed");
        }
    }
}

public class Program {
    static void Main(string[] args) {
        IContainer container = IoCBuilder.Build();

        Order customerOrder = new Order { Id = 1, ProductName = "Product", Quantity = 3, UnitPrice = 75 };

        Commerce commerce = container.Resolve<Commerce>();

        commerce.ProcessOrder(customerOrder);
    }
}
```

# Referenties

- <https://martinfowler.com/articles/injection.html>
- <https://www.c-sharpcorner.com/article/dependency-injection-day-1/>
- <https://www.c-sharpcorner.com/article/dependency-injection-part-2-5-reasons-to-write-loosely-coupled-code/>
- <https://www.c-sharpcorner.com/article/dependency-injection-part-3/>
- <https://www.c-sharpcorner.com/article/dependency-injection-part-four-embracing-abstraction/>
- <https://www.c-sharpcorner.com/article/dependency-injection-part-5-using-a-di-container-autofac/>

# BUILDER

---

# Builder: concepten

- Builderpatroon bestaat uit 4 componenten:
  - Builder interface: template die de stappen beschrijft om product te maken
  - Concrete Builder: implementeert de Builder interface en voorziet een interface om product te accessen
  - Director: maakt het object d.m.v. Builder interface
  - Product: eindproduct dat gemaakt wordt
- Voorbeeld van bier brouwen
- Klasse Beer => product dat gemaakt zal worden
- IBeerBuilder interface => beschrijft de stappen die nodig zijn om beer te brouwen  
+ GetBeer methode dat Beer object teruggeeft

# Klasse Beer (product)

```
public class Beer {  
    public string Label { get; set; }  
    public double Volume { get; set; }  
    public double Price { get; set; }  
    public double Potency { get; set; }  
  
    public override string ToString() {  
        return String.Format("{0} {1} oz. {2}% ABV  
{3:C}", Label, Volume, Potency, Price);  
    }  
}
```

# IBeerBuilder interface (IBuilder interface)

```
public interface IBeerBuilder {  
    void Brew();  
    void Ferment();  
    void Bottle();  
    void Age();  
    Beer GetBeer();  
}
```

# Concrete builders (2 klassen)

```
class AmberAleBuilder : IBeerBuilder {  
    private Beer _beer = new Beer();  
  
    public void Brew() {  
        _beer.Volume = 64;  
    }  
  
    public void Ferment() {  
        _beer.Potency = 5.5;  
    }  
}
```

# Concrete builders (2 klassen)

```
public void Bottle() {  
    _beer.Label = "Amber Ale";  
}  
  
public void Age() {  
    _beer.Price = 16;  
}  
  
public Beer GetBeer() {  
    return _beer;  
}  
}
```



# Concrete builders (2 klassen)

```
class StoutBuilder : IBeerBuilder {  
    private Beer _beer = new Beer();  
  
    public void Brew() {  
        _beer.Volume = 72;  
    }  
  
    public void Ferment() {  
        _beer.Potency = 6.5;  
    }  
}
```

# Concrete builders (2 klassen)

```
public void Bottle() {  
    _beer.Label = "Sturdy Stout";  
}  
  
public void Age() {  
    _beer.Price = 22;  
}  
  
public Beer GetBeer() {  
    return _beer;  
}  
}
```

# Klasse Director

```
public class BrewMaster {  
    IBeerBuilder _beerBuilder;  
  
    public void SetBeerBuilder(IBeerBuilder beerBuilder) {  
        _beerBuilder = beerBuilder;  
    }  
  
    public Beer GetBeer() {  
        return _beerBuilder.GetBeer();  
    }  
}
```

# Klasse Director

```
    public void BrewBeer() {  
        _beerBuilder.Brew();  
        _beerBuilder.Ferment();  
        _beerBuilder.Bottle();  
        _beerBuilder.Age();  
    }  
}
```

# Alles samenvoegen

```
public static void Main(string[] args) {  
    BrewMaster brewMeister = new BrewMaster();  
    brewMeister.SetBeerBuilder(new AmberAleBuilder());  
    brewMeister.BrewBeer();  
    Beer amberAle = brewMeister.GetBeer();  
    Console.WriteLine(amberAle);  
    brewMeister.SetBeerBuilder(new StoutBuilder());  
    brewMeister.BrewBeer();  
    Beer stout = brewMeister.GetBeer();  
    Console.WriteLine(stout);  
    Console.ReadLine();  
}
```

# Builder: conclusie

- Sterkte van patroon
  - Opbreken in kleinere delen van constructie van een complex object
  - Je kan constructieproces wegstoppen voor de klant
  - Moedigt separation of concerns aan & uitbreidbaarheid van de applicatie

# Referenties Builderpatroon

- Zie <http://blogs.tedneward.com/patterns/Builder-CSharp/>
- <https://visualstudiomagazine.com/articles/2012/07/16/the-builder-pattern-in-net.aspx>

# STRATEGY & FACTORY

---



# Strategy & Factory pattern

- Stel je werkt voor een bank en je moet een programma schrijven om **maandelijkse** interest van verschillende soorten rekeningen te bepalen: bijv. zichtrekening, spaarrekening, ...
- Je begint met het maken van een enum

```
public enum AccountTypes { CURRENT, SAVINGS}
```

- En je schrijft volgende klasse

# Klasse InterestCalculator (eerste poging)

```
public class InterestCalculator {  
    public double calculateInterest(AccountTypes accountType, double  
                                   accountBalance) {  
        switch (accountType) {  
            case CURRENT: return accountBalance * (0.02 / 12);  
            case SAVINGS: return accountBalance * (0.04 / 12);  
            default:  
                return 0;  
        }  
    }  
}
```

# Uitbreidbaarheid?

- Stel je wilt extra rekeningen toevoegen om daarvan maandelijkse rente te berekenen => wordt complex

# Klasse InterestCalculator (eerste poging)

```
public class InterestCalculator {  
    public double calculateInterest(AccountTypes accountType, double accountBalance) {  
        switch (accountType) {  
            case CURRENT: return accountBalance * (0.02 / 12);  
            case SAVINGS: return accountBalance * (0.04 / 12);  
            case STANDARD_MONEY_MARKET: return accountBalance * (0.06/12);  
            case HIGH_ROLLER_MONEY_MARKET: return accountBalance < 100000.00  
                                                ? 0 : accountBalance * (0.075/12);  
            default:  
                return 0;  
        }  
    }  
}
```

# Oplossing: strategy en factory patroon

```
public interface InterestCalculationStrategy{  
    double calculateInterest(double accountBalance);  
}
```

# Implementaties van interface

```
public class CurrentAccountInterestCalculation : InterestCalculationStrategy {  
    public double calculateInterest(double accountBalance) {  
        return accountBalance * (0.02 / 12);  
    }  
}  
  
public class SavingsAccountInterestCalculation : InterestCalculationStrategy {  
    public double calculateInterest(double accountBalance) {  
        return accountBalance * (0.04 / 12);  
    }  
}
```

# Implementaties van interface

```
public class MoneyMarketInterestCalculation : InterestCalculationStrategy {  
    public double calculateInterest(double accountBalance) {  
        return accountBalance * (0.06 / 12);  
    }  
}
```

```
public class HighRollerMoneyMarketInterestCalculation : InterestCalculationStrategy {  
    public double calculateInterest(double accountBalance) {  
        return accountBalance < 1000000.0 ? 0 : accountBalance * (0.075 / 12);  
    }  
}
```

# Factorypatroon toepassen

```
public class InterestCalculationStrategyFactory {  
    private InterestCalculationStrategy  
currentAccountInterestCalculationStrategy = new CurrentAccountInterestCalculation();  
  
    private InterestCalculationStrategy  
savingsAccountInterestCalculationStrategy = new SavingsAccountInterestCalculation();  
  
    private InterestCalculationStrategy  
moneyMarketAccountInterestCalculationStrategy = new  
MoneyMarketInterestCalculation();  
  
    private InterestCalculationStrategy  
highRollerMoneyMarketAccountInterestCalculationStrategy = new  
HighRollerMoneyMarketInterestCalculation();  
  
    private InterestCalculationStrategy noInterestCalculationStrategy = new  
NoInterestCalculation();  
}
```



# Factorypatroon toepassen

```
public InterestCalculationStrategy GetInterestCalculationStrategy
                                   (AccountTypes accountType) {

    switch (accountType) {
        case AccountTypes.CURRENT:
            return currentAccountInterestCalculationStrategy;

        case AccountTypes.SAVINGS:
            return savingsAccountInterestCalculationStrategy;

        case AccountTypes.STANDARD_MONEY_MARKET:
            return moneyMarketAccountInterestCalculationStrategy;

        case AccountTypes.HIGH_ROLLER_MONEY_MARKET:
            return highRollerMoneyMarketAccountInterestCalculationStrategy;

        default: return noInterestCalculationStrategy;
    }
}
```

}

# Onze oorspronkelijke InterestCalculator wordt

```
public class InterestCalculator {  
    private InterestCalculationStrategyFactory  
interestCalculationStrategyFactory = new InterestCalculationStrategyFactory();  
    public double CalculateInterest(AccountTypes accountType, double  
                                accountBalance) {  
        InterestCalculationStrategy interestCalculationStrategy =  
interestCalculationStrategyFactory.GetInterestCalculationStrategy(accountType);  
        return interestCalculationStrategy.calculateInterest(accountBalance);  
    }  
}
```

# Referenties: Strategy en factory patroon

- Zie <https://dzone.com/articles/design-patterns-the-strategy-and-factory-patterns>

# **SOLID PRINCIPLES**

---

# S.O.L.I.D. principles

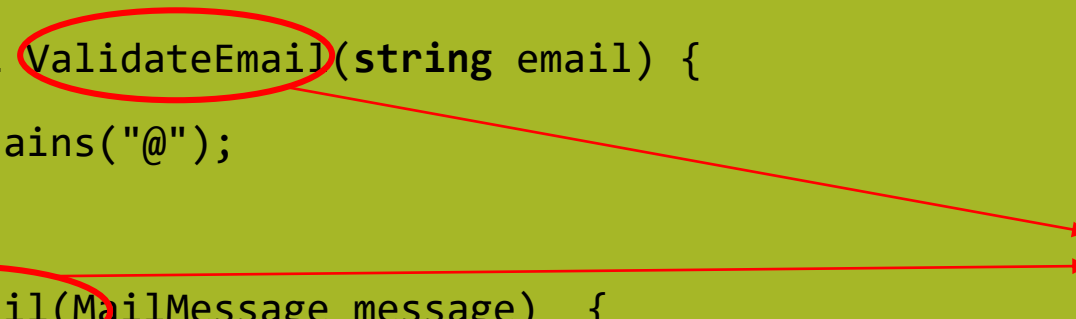
- S: **S**ingle **R**esponsibility **P**inciple (SRP)
  - O: **O**pen **C**losed **P**inciple (OCP)
  - L: **L**iskov **S**ubstitution **P**inciple (LSP)
  - I: **I**nterface **S**egregation **P**inciple (ISP)
  - D: **D**ependency **I**nversion **P**inciple (DIP)
- 
- Het volgen van deze principes vermijdt veel van de meest voorkomende problemen bij het ontwikkelen en onderhouden van applicaties
  - Van tightly coupled naar loosely coupled

# Single Responsibility Principle

- SRP: “every software module should have only one reason to change”
- Betekenis: elke klasse zou eigenlijk maar één job moeten doen. Alles in de klasse moet gerelateerd zijn aan één enkel doel
- Dus: geen Zwitsers zakmes waar één ding in klasse veranderen invloed heeft op ganse applicatie
- Betekent dus niet dat er maar één methode mag zijn in klasse! Er mogen verschillende leden in de klasse zijn, zolang ze maar gerelateerd zijn aan één enkele verantwoordelijkheid

# SRP: hoe het niet moet

```
public class UserService {  
    public void Register(string email, string password) {  
        if (!ValidateEmail(email))  
            throw new ValidationException("Email is not an email");  
        var user = new User(email, password);  
        SendEmail(new MailMessage("mysite@nowhere.com", email) { Subject="Hello fool" });  
    }  
    public virtual bool ValidateEmail(string email) {  
        return email.Contains("@");  
    }  
    public bool SendEmail(MailMessage message) {  
        _smtpClient.Send(message);  
    }  
}
```



Hebben eigenlijk niet  
veel te zien met de klasse  
UserService

# SRP: hoe het beter kan

```
public class UserService {  
    EmailService _emailService;  
    DbContext _dbContext;  
  
    public UserService(EmailService aEmailService, DbContext aDbContext) {  
        _emailService = aEmailService;  
        _dbContext = aDbContext;  
    }  
  
    public void Register(string email, string password) {  
        if (!_emailService.ValidateEmail(email))  
            throw new ValidationException("Email is not an email");  
        var user = new User(email, password);  
        _dbContext.Save(user);  
        emailService.SendEmail(new MailMessage("myname@mydomain.com", email) {Subject="Hello fool!"});  
    }  
}
```



# SRP: hoe het beter kan

```
public class EmailService {  
    SmtplibClient _smtpClient;  
    public EmailService(SmtplibClient aSmtplibClient) {  
        _smtpClient = aSmtplibClient;  
    }  
    public bool virtual ValidateEmail(string email) {  
        return email.Contains("@");  
    }  
    public bool SendEmail(MailMessage message) {  
        _smtpClient.Send(message);  
    }  
}
```

# Open/Closed Principle

- OCP: "A software module/class is open for extension and closed for modification"
- Closed for modifications: klasse is in principe grondig getest (unit testing) en het is niet de bedoeling om geteste klasse aan te passen (tenzij we bugs zouden vinden).
- Anderzijds moet een klasse wel open zijn voor extension: bijv. door overerving

# OCP: hoe het niet moet

```
public class Rectangle{
    public double Height {get;set;}
    public double Width {get;set; }
}

public class AreaCalculator {
    public double TotalArea(Rectangle[] arrRectangles) {
        double area;
        foreach(var objRectangle in arrRectangles) {
            area += objRectangle.Height * objRectangle.Width;
        }
        return area;
    }
}
```

# OCP

- Stel: je wilt ook de oppervlakte van andere figuren uitrekenen (cirkel, driehoek,...)
- Probleem met de methode TotalArea

```
public class Rectangle{  
    public double Height {get;set;}  
    public double Width {get;set; }  
}  
  
public class Circle{  
    public double Radius {get;set;}  
}
```

```
public class AreaCalculator {  
    public double TotalArea(object[] arrObjects) {  
        double area = 0;  
        Rectangle objRectangle;  
        Circle objCircle;  
        foreach(var obj in arrObjects) {  
            if(obj is Rectangle) {  
                objRectangle = (Rectangle)obj;  
                area += obj.Height * obj.Width;  
            }  
            else {  
                objCircle = (Circle)obj;  
                area += objCircle.Radius * objCircle.Radius * Math.PI;  
            }  
        }  
        return area;  
    }  
}
```

# Maar ...

- Wat als we oppervlakte van een driehoek willen berekenen?
- Methode TotalArea aanpassen.
- => klasse AreaCalculator is niet gesloten voor aanpassingen (not closed for modification!)
- Design aanpassen zodat dit wel het geval is

# DuS...

```
public abstract class Shape {  
    public abstract double Area();  
}  
public class Rectangle: Shape {  
    public double Height {get;set;}  
    public double Width {get;set;}  
    public override double Area() {  
        return Height * Width;  
    }  
}
```

```
public class Circle: Shape {  
    public double Radius {get;set;}  
    public override double Area() {  
        return Radius * Radius * Math.PI;  
    }  
}
```

# Uiteindelijk: methode TotalArea

```
public class AreaCalculator {  
    public double TotalArea(Shape[] arrShapes) {  
        double area=0;  
        foreach(var objShape in arrShapes) {  
            area += objShape.Area();  
        }  
        return area;  
    }  
}
```

- Klasse  
AreaCalculator  
hoeft niet meer  
aangepast te  
worden als er een  
nieuwe vorm  
geïntroduceerd  
wordt



# Liskov Substitution Principle

- LSP: “you should be able to use any derived class instead of a parent class and have it behave in the same manner without modification”
- Een subklasse mag het gedrag van de superklasse niet beïnvloeden

# LSP zoals het niet hoort

```
public class SqlFile {  
    public string FilePath {get;set;}  
    public string FileText {get;set;}  
    public string LoadText() {  
        /* Code to read text from sql file */  
    }  
    public string SaveText() {  
        /* Code to save text into sql file */  
    }  
}
```

# LSP zoals het niet hoort

```
public class SqlFileManager {  
    public List<SqlFile> lstSqlFiles {get;set}  
    public string GetTextFromFiles() {  
        StringBuilder objStrBuilder = new StringBuilder();  
        foreach(var objFile in lstSqlFiles) {  
            objStrBuilder.Append(objFile.LoadText());  
        }  
        return objStrBuilder.ToString();  
    }  
    public void SaveTextIntoFiles() {  
        foreach(var objFile in lstSqlFiles) {  
            objFile.SaveText();  
        }  
    }  
}
```

# LSP zoals het niet hoort

- Stel nu dat er gevraagd wordt om rekening te houden met read-only bestanden in een bepaalde folder.
- Dit betekent dat je in deze bestanden niets mag opslaan

```
public class SqlFile {  
    public string LoadText() {  
        /* Code to read text from sql file */  
    }  
    public void SaveText() {  
        /* Code to save text into sql file */  
    }  
}
```

# LSP zoals het niet hoort

```
public class ReadOnlySqlFile: SqlFile {  
    public string FilePath {get;set;}  
    public string FileText {get;set;}  
    public string LoadText() {  
        /* Code to read text from sql file */  
    }  
    public void SaveText() {  
        /* Throw an exception when app flow tries to do save. */  
        throw new IOException("Can't Save");  
    }  
}
```

# LSP zoals het niet hoort

```
public class SqlFileManager {  
    public List<SqlFile> lstSqlFiles {get;set}  
    public string GetTextFromFiles() {  
        StringBuilder objStringBuilder = new StringBuilder();  
        foreach(var objFile in lstSqlFiles) {  
            objStringBuilder.Append(objFile.LoadText());  
        }  
        return objStringBuilder.ToString();  
    }  
}
```

# LSP zoals het niet hoort

```
public void SaveTextIntoFiles() {  
    foreach(var objFile in lstSqlFiles) {  
        //Check whether the current file object is read only or not.If yes, skip calling it's  
        // SaveText() method to skip the exception.  
        if(! objFile is ReadOnlySqlFile)  
            objFile.SaveText();  
    }  
}  
}
```

# Probleem

- We hebben in SqlFileManager de methode SaveTextIntoFiles() aangepast
- We kunnen ReadOnlySqlFile klasse niet als vervanger gebruiken van zijn parent zonder SqlFileManager aan te passen
  - Niet ontworpen volgens LSP
- Hoe LSP maken?
  - Gebruik maken van interfaces



# LSP zoals het hoort

```
public interface IReadableSqlFile {  
    string LoadText();  
}  
  
public interface IWritableSqlFile {  
    void SaveText();  
}
```

```
public class ReadOnlySqlFile: IReadableSqlFile {  
  
    public string FilePath {get;set;}  
    public string FileText {get;set;}  
    public string LoadText() {  
        /* Code to read text from sql file */  
    }  
}
```

# LSP zoals het hoort

```
public class SqlFile: IWritableSqlFile, IReadableSqlFile {  
    public string FilePath {get;set;}  
    public string FileText {get;set;}  
    public string LoadText() {  
        /* Code to read text from sql file */  
    }  
    public void SaveText() {  
        /* Code to save text into sql file */  
    }  
}
```

# LSP zoals het hoort

```
public class SqlFileManager {  
    public string GetTextFromFiles(List<IReadableSqlFile> aLstReadableFiles) {  
        StringBuilder objStrBuilder = new StringBuilder();  
        foreach(var objFile in aLstReadableFiles) {  
            objStrBuilder.Append(objFile.LoadText());  
        }  
        return objStrBuilder.ToString();  
    }  
    public void SaveTextIntoFiles(List<IWritableSqlFile> aLstWritableFiles) {  
        foreach(var objFile in aLstWritableFiles) {  
            objFile.SaveText();  
        }  
    }  
}
```

# Interface Segregation Principle

- ISP: “clients should not be forced to implement interfaces they don’t use. Instead of one fat interface many small interfaces are preferred based on groups of methods, each one serving one sub module.”

# Codevoorbeeld dat niet voldoet aan ISP

- Stel: applicatie bouwen voor een IT-bedrijf
- Bevat rollen zoals
  - TeamLead: verdeelt grotere taken in kleinere taken en geeft die aan Programmers of voert het zelf uit
  - Programmer: voert taken uit
  - Manager: wijst taken toe aan TeamLead, maar voert zelf geen taken uit

# Codevoorbeeld dat niet voldoet aan ISP

```
public Interface ILead {  
  
    void CreateSubTask();  
    void AssignTask();  
    void WorkOnTask();  
}  
  
public class TeamLead : ILead {  
  
    public void AssignTask() {  
        //Code to assign a task.  
    }  
  
    public void CreateSubTask() {  
        //Code to create a sub task  
    }  
  
    public void WorkOnTask() {  
        //Code to implement perform assigned task.  
    }  
}
```

```
public class Manager: ILead {  
  
    public void AssignTask() {  
        //Code to assign a task.  
    }  
  
    public void CreateSubTask() {  
        //Code to create a sub task.  
    }  
  
    public void WorkOnTask() {  
        throw new Exception("Manager can  
                               't work on Task");  
    }  
}
```

# Codevoorbeeld volgens ISP

```
public interface IProgrammer
{
    void WorkOnTask();
}

public interface ILead {
    void AssignTask();
    void CreateSubTask();
}
```

```
public class Programmer: IProgrammer {
    public void WorkOnTask() {
        //code to implement to work on the Task.
    }
}

public class Manager: ILead {
    public void AssignTask() {
        //Code to assign a Task
    }

    public void CreateSubTask() {
        //Code to create a sub taks from a task.
    }
}
```

# Codevoorbeeld volgens ISP

```
public class TeamLead: IProgrammer, ILead {  
    public void AssignTask() {  
        //Code to assign a Task  
    }  
    public void CreateSubTask() {  
        //Code to create a sub task from a task.  
    }  
    public void WorkOnTask() {  
        //code to implement to work on the Task.  
    }  
}
```



# Dependency Inversion Principle

- “High-level modules/classes should not depend upon low-level modules/classes. Both should depend on abstractions. Secondly, abstractions should not depend upon details. Details should depend upon abstractions.”
- High-level modules/classes:
  - Implementeren business logic
- Low-level modules/classes:
  - E.g. schrijven van data naar databank,...
- High-level module die afhangt van low-level module, en die veel weet over de klassen waarmee het interageert is “tightly coupled”
- Wanneer een klasse veel weet over implementatie van een andere klasse:
  - Risico bestaat dat verandering aan ene klasse de andere klasse “breekt”
- Dus: zoveel mogelijk: loosely coupled klassen

# DI zoals het niet moet

- Stel:
  - Error logging module, die uitzonderingen logt in een bestand

```
public class FileLogger {  
    public void LogMessage(string aStackTrace) {  
        //code to log stack trace into a file.  
    }  
}
```

# DI zoals het niet moet

```
public static class ExceptionLogger {  
    public static void LogIntoFile(Exception aException) {  
        FileLogger objFileLogger = new FileLogger();  
        objFileLogger.LogMessage(GetUserReadableMessage(aException));  
    }  
    private string GetUserReadableMessage(Exception ex) {  
        string strMessage = string.Empty;  
        //code to convert Exception's stack trace and message to user readable format.  
        return strMessage;  
    }  
}
```

# DI zoals het niet hoort

Client code:

```
public class DataExporter {  
    public void ExportDataFromFile() {  
        try {  
            //code to export data from files to database.  
        }  
        catch(Exception ex) {  
            new ExceptionLogger().LogIntoFile(ex);  
        }  
    }  
}
```

# DI zoals het niet hoort

- Stel nu dat client de IO Exceptions in een databank opslaat, de andere excepties worden in bestand opgeslaan.

```
public class DbLogger {  
    public void LogMessage(string aMessage) {  
        //Code to write message in database.  
    }  
}  
  
public class FileLogger {  
    public void LogMessage(string aStackTrace) {  
        //code to log stack trace into a file.  
    }  
}
```

# DI zoals het niet hoort

```
public class ExceptionLogger {  
    public void LogIntoFile(Exception aException) {  
        FileLogger objFileLogger = new FileLogger();  
        objFileLogger.LogMessage(GetUserReadableMessage(aException));  
    }  
    public void LogIntoDataBase(Exception aException) {  
        DbLogger objDbLogger = new DbLogger();  
        objDbLogger.LogMessage(GetUserReadableMessage(aException));  
    }  
    private string GetUserReadableMessage(Exception ex) {  
        string strMessage = string.Empty;  
        //code to convert Exception's stack trace and message to user readable format.  
        return strMessage;  
    }  
}
```

# DI zoals het niet hoort

Client code:

```
public class DataExporter {  
    public void ExportDataFromFile() {  
        try {  
            //code to export data from files to database.  
        }  
        catch(IOException ex) {  
            new ExceptionLogger().LogIntoDataBase(ex);  
        }  
        catch(Exception ex) {  
            new ExceptionLogger().LogIntoFile(ex);  
        }  
    }  
}
```

# DI zoals het niet hoort: oplossing

- Wat gebeurt als clients nog een nieuwe logger willen?
- In ExceptionLogger moet er een extra methode toegevoegd worden
- De laatste klasse wordt alsmaar groter, telkens de clients een nieuwe logger willen
- Waarom gebeurt dit?
  - ExceptionLogger is direct verbonden met de low-level klassen FileLogger en DbLogger
- Design van klasse aanpassen!
  - Zodat ExceptionLogger loosely coupled is
- Abstractie toevoegen



# DI zoals het moet

```
public interface ILogger {  
    public void LogMessage(string aString);  
}  
  
public class DbLogger: ILogger {  
    public void LogMessage(string aMessage) {  
        //Code to write message in database.  
    }  
}  
  
public class FileLogger: ILogger {  
    public void LogMessage(string aStackTrace) {  
        //code to log stack trace into a file.  
    }  
}
```

# DI zoals het moet

```
public class ExceptionLogger {
    private ILogger _logger;
    public ExceptionLogger(ILogger aLogger) {
        this._logger = aLogger;
    }
    public void LogException(Exception aException) {
        string strMessage = GetUserReadableMessage(aException);
        this._logger.LogMessage(strMessage);
    }
    private string GetUserReadableMessage(Exception aException) {
        string strMessage = string.Empty;
        //code to convert Exception's stack trace and message to user readable format.
        return strMessage;
    }
}
```

# DI zoals het moet

```
public class DataExporter {  
    public void ExportDataFromFile() {  
        ExceptionLogger _exceptionLogger;  
        try {  
            //code to export data from files to database.  
        }  
        catch(IOException ex) {  
            _exceptionLogger = new ExceptionLogger(new DbLogger());  
            _exceptionLogger.LogException(ex);  
        }  
        catch(Exception ex) {  
            _exceptionLogger = new ExceptionLogger(new FileLogger());  
            _exceptionLogger.LogException(ex);  
        }  
    }  
}
```

# DI zoals het moet

- Stel dat we een nieuwe logger wensen toe te voegen

```
public class EventLogger: ILogger {  
    public void LogMessage(string aMessage) {  
        //Code to write message in system's event viewer.  
    }  
}
```

# DI zoals het moet

```
public class DataExporter {  
    public void ExportDataFromFile() {  
        ExceptionLogger _exceptionLogger;  
        try {  
            //code to export data from files to database.  
        }  
        catch(IOException ex) {  
            _exceptionLogger = new ExceptionLogger(new DbLogger());  
            _exceptionLogger.LogException(ex);  
        }  
    }  
}
```

# DI zoals het moet

```
catch(SQLException ex) {  
    _exceptionLogger = new ExceptionLogger(new EventLogger());  
    _exceptionLogger.LogException(ex);  
}  
catch(Exception ex) {  
    _exceptionLogger = new ExceptionLogger(new FileLogger());  
    _exceptionLogger.LogException(ex);  
}  
}  
}
```

# Referenties

- Damodhar Naidu: Solid Principle in C# (<http://www.c-sharpcorner.com/UploadFile/damubetha/solid-principles-in-C-Sharp/>)