Het Factory Pattern

Analysis & design 2

Pattern of patterns?

Er is niet 1 factory pattern.

Er zijn 2,5 verschillende factory patterns.

- Simple Factory (telt maar half volgens de kenners)
- Factory Method
- Abstract Factory

Waarom factory patterns?

Factory pattern: doel

Alle factory patterns houden zich bezig met de *creatie van objecten*.

Ze proberen de beperkingen opgelegd door new() te verminderen.


YOUR NEW IS BAD
AND YOU SHOULD FEEL BAD
quickmeme.com

new is bad?

```
Gearbox gearbox = new ManualGearbox();
```

Dit type kan iets abstract zijn…

Maar dit type moet iets concreet zijn!

Een constructor gebruiken betekent je vastpinnen op een concrete klasse.

Die afhankelijkheid is niet altijd wenselijk.

More new is even worse

```
// Concreet gearboxtype pas at runtime gekend
Gearbox gearbox = null;
if (typeWanted == "manual")
    gearbox = new ManualGearbox();
else if (typeWanted == "automatic")
    gearbox = new AutomaticGearbox();
```

Elke client die een gearbox nodig heeft moet dezelfde if-else gebruiken.

Als er een derde type gearbox bijkomt, dan moet elke client die if-else aanpassen.

Simple Factory

Creatie van objecten

Het Simple Factory (pseudo-)pattern

Oplossing voor dit probleem:

isoleer wat verandert.

Dus: stop de if-else in een aparte klasse.

Die klasse wordt verantwoordelijk voor het creëren van gearboxen: een factory-klasse.

Dit is het **Simple Factory pattern**.

De factoryklasse

```
public class GearboxFactory {
    public enum GearboxType {Automatic, Manual};

    public Gearbox createGearbox(GearboxType type) {
        if (type == GearboxType.Automatic)
            return new AutomaticGearbox();
        else if (type == GearboxType.Manual)
            return new ManualGearbox();
        else
            throw new ArgumentException("Unknown gearbox type");
    }
}
```

De clientcode

```
GearboxFactory gearboxFactory = new GearboxFactory();
Gearbox gearbox =
 gearboxFactory.createGearbox(
                    GearboxFactory.GearboxType.Automatic
 );
```

Variaties

- Meestal is het mogelijk om de create-methode static te maken. Dan moet je de factoryklasse niet meer instantiëren.

- Dit pattern kan ook als de factory maar 1 concreet type maakt (dus zonder if-else in de create-methode).
  In dat geval kun je eventueel de factoryklasse weglaten. De create-methode wordt dan een static methode van de productklasse.

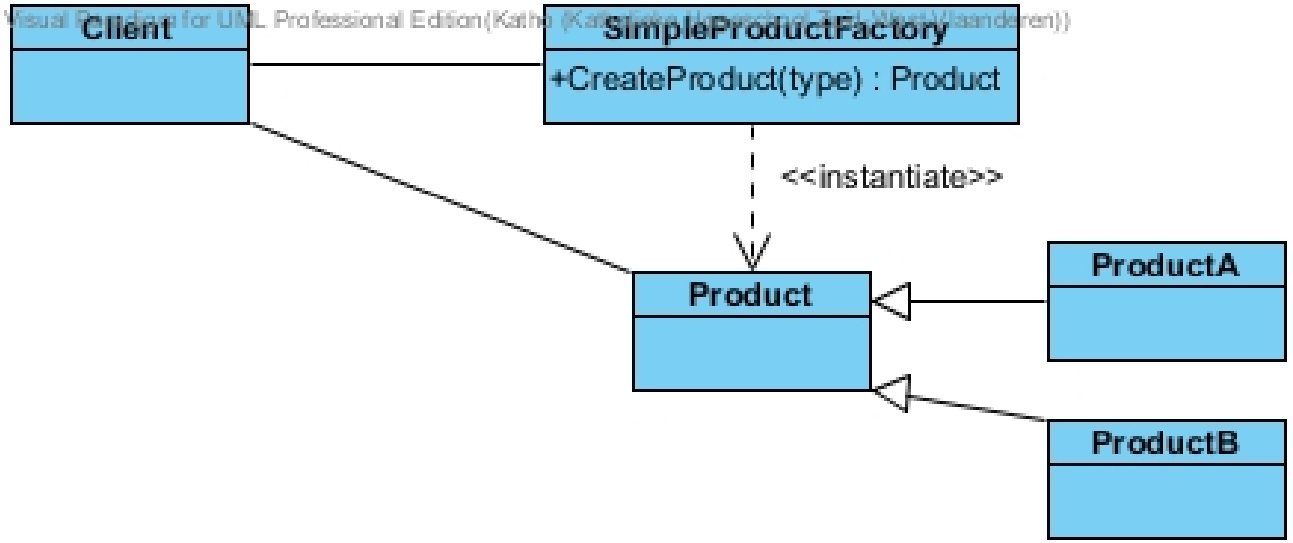
Simple factory pattern
Samenvatting

- Simple factory gebruik je voor de creatie van objecten uit een familie van objecten.

- De verantwoordelijkheid voor de creatie wordt gegeven aan een factoryklasse.

- Zo zijn clients niet meer afhankelijk van de concrete types van de objecten die gemaakt worden (*dependency inversion*).

Simple factory pattern
Schema



Factory Method pattern

Creatie van objecten, als verschillende clients verschillende behoeften hebben.

Het Factory Method pattern

Bij Simple Factory is er één factory voor alle clienten.

Het Factory Method pattern laat toe om de factories te specialiseren.

Er worden nog steeds objecten van één familie gemaakt, maar er zijn verschillende factories die elk aangepast zijn aan de behoefte van een bepaald soort client.

Factory Method pattern voorbeeld

Voorbeeld:

Clients hebben voertuigen nodig in drie soorten: economical, midrange en powerful.

Maar de ene client moet wagens hebben, en de andere bestelwagens.

Er is dus een stuk gemeenschappelijke behoefte: de voertuigen moeten economical, midrange of powerful zijn.

Anderzijds heeft elke client een specifieke behoefte: wagens of bestelwagens.

Factory Method pattern aanpak

Hoe pakt het Factory Method pattern dit aan?

- Er is een familie van factory klassen.
- Het gemeenschappelijke gedrag zit in de basisklasse.
- Het gedrag dat specifiek is voor een bepaalde client zit in een afgeleide klasse.

De basisklasse van de factory familie

```
public abstract class VehicleFactory {
  public enum DrivingStyle {Economical, Midrange, Powerful};

    // het gemeenschappelijk gedrag
    public Vehicle build(DrivingStyle style, Vehicle.Colour colour)
    {
      // concrete creatie gebeurt door de factory method
        Vehicle v = selectVehicle(style);
        v.paint(colour);
        return v;
    }

    // het gespecialiseerde gedrag
    protected abstract Vehicle selectVehicle(DrivingStyle style);
}
```

De basisklasse

Merk op:

- De basisklasse is abstract.
- Methode selectVehicle is abstract.
- De methode build() is public, maar selectVehicle (de factory method) is protected.
  Het is niet de bedoeling dat clients rechtstreeks selectVehicle aanroepen.
- Het returntype van build() is een type dat voor alle clients bruikbaar is.

Een concrete factory

```
public class CarFactory extends VehicleFactory {
  protected Vehicle selectVehicle(
                          DrivingStyle style) {
     if (style == DrivingStyle.Economical)
        return new Saloon();
     else if (style == DrivingStyle.Midrange)
        return new Coupe();
     else
        return new Sport();
   }
}
```

Concrete factory

Merk op:

- De concrete factory is afgeleid van de basisklasse.
- De concrete factory heeft een override van de factory method selectVehicle.

Abstract Factory pattern

Creatie van bij elkaar horende types.

Het Abstract Factory pattern

Factory pattern nummer 2,5:

Abstract Factory

Het Abstract Factory pattern

Het creatieprobleem wordt uitgebreid: clients hebben niet één type objecten nodig, maar meerdere types.

Bovendien hebben verschillende clients verschillende concrete behoeften.

Abstract Factory pattern voorbeeld

Voor het maken van een auto heb je een chassis, een carrosserie en een stel ramen nodig.

Elk type auto heeft een eigen type chassis, een eigen type carrosserie, en een eigen type ramen.
Welke concrete chassis, carrosserie en ramen nodig zijn hangt dus af van het type auto dat gemaakt moet worden. Er kan niet zomaar gemengd worden.

Abstract Factory pattern aanpak

Het Abstract Factory pattern gebruikt factory methods, zoals in het Factory Method pattern.

Net als bij het Factory Method pattern is er een familie van factoryklassen.

Elke concrete factoryklasse is verantwoordelijk voor het maken van de onderdelen van één type auto.

Het gemeenschappelijke zit in de basisklasse.
Wat is er hier gemeenschappelijk? Eigenlijk niet zo veel: enkel het feit dat een auto een chassis, een carrosserie en ramen nodig heeft.

Abstract Factory pattern
de basisklasse

```
// Basisklasse voor de onderdelenfactories
public abstract class AbstractVehicleFactory {
  public abstract Body createBody();
  public abstract Chassis createChassis();
  public abstract Windows createWindows();
}
```

Merk op:

- Dit is een zuiver abstracte klasse. Het had ook een interface kunnen zijn.

Abstract Factory pattern
een concrete klasse

```
// Concrete onderdelenfactory
public class CarFactory extends AbstractVehicleFactory {
  public Body createBody() {
    return new CarBody();
  }
  public Chassis createChassis() {
    return new CarChassis();
  }
  public Windows createWindows() {
    return new CarWindows();
  }
}
```

Het abstract factory pattern
Samengevat

De **Client** heeft **Producten** nodig, en krijgt die van de **Factory**.

Om niet afhankelijk te zijn van concrete Producten, gebruikt de Client **Product-interfaces**.

Om niet afhankelijk te zijn van concrete Factories, gebruikt de Client een **Factory-interface: de Abstract Factory**.

Een **Factory** is een implementatie van de Factory-interface.

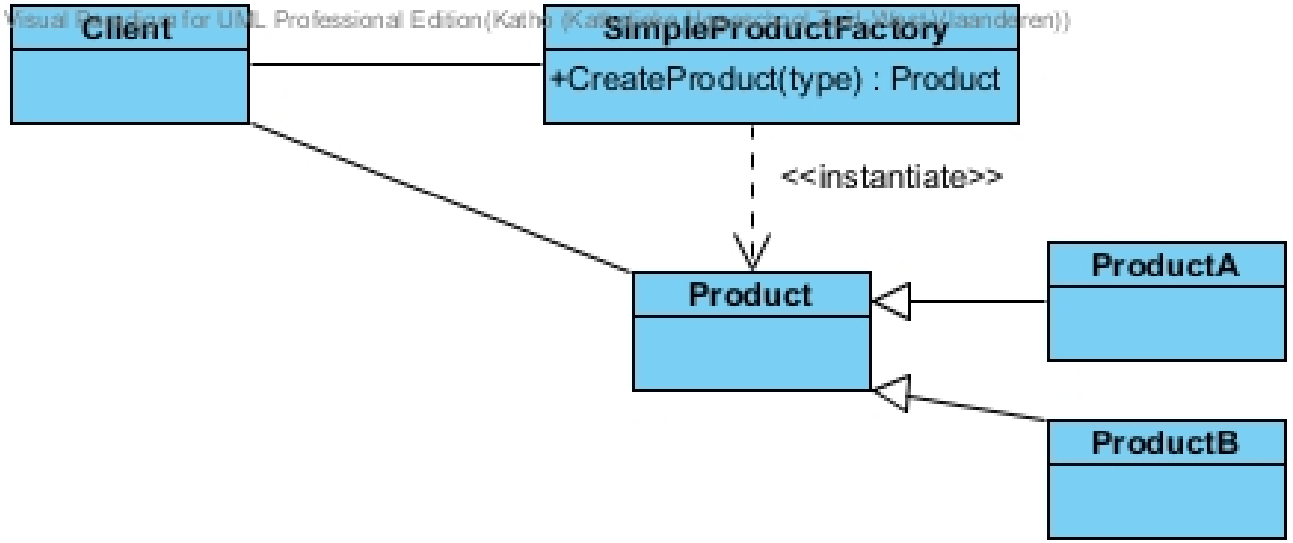
Een Factory **creëert** concrete **Producten**.

Een **Product** is een implementatie van de Product-interface.

Nog eens alle types factory pattern op een rijtje.

Nog eens alle factory-varianten op een rijtje

1. Simple factory



- Creatie van een soort Product
- Client beslist welk type
- Client is enkel afhankelijk van superklasse Product, niet van ProductA of ProductB.

Nog eens alle factory-varianten op een rijtje

2. Factory Method

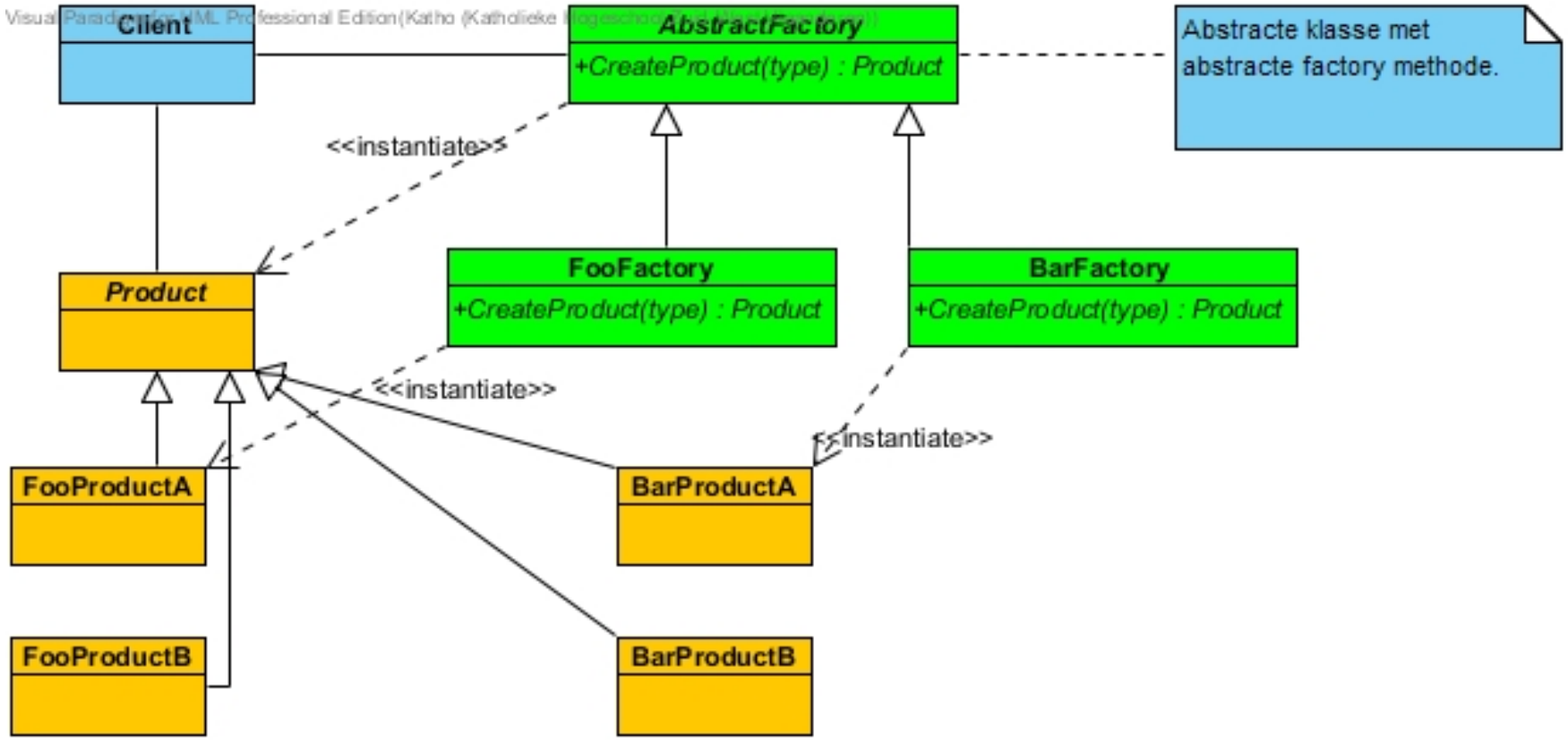
Simple factory, maar nu met meerdere soorten factories!

De client beslist at runtime welke factory er gebruikt wordt.

Alle gemeenschappelijk gedrag van de factories wordt samengebracht in een abstracte klasse.

Alle concrete factories zijn afgeleid van die abstracte klasse.

Nog eens alle factory-varianten op een rijtje
Factory method pattern: schema



Nog eens alle factory-varianten op een rijtje

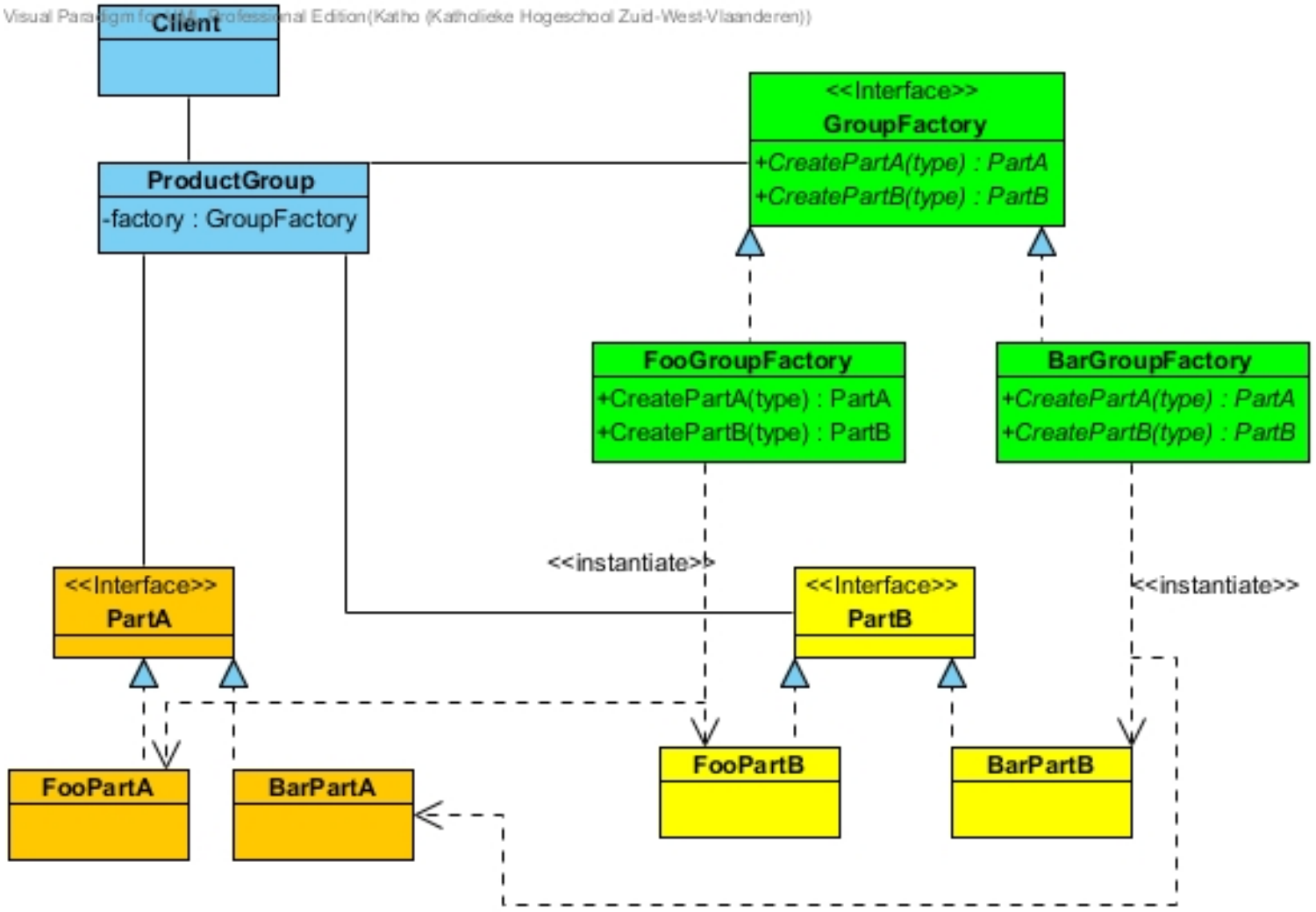
3. Abstract Factory

Simple factory, maar nu met een groep van producten! (en dus met meerdere factory methoden per factory).

De soorten producten kunnen niet willekeurig samengegooid worden; de creatie moet gecoördineerd worden.

Typisch voor dit pattern: een groep heeft zijn eigen factory (*compositie*).

Nog eens alle factory-varianten op een rijtje
Abstract factory pattern: schema



Tot slot

De naamgeving van al die patterns lijkt wat ongelukkig…

- Het Simple Factory pattern heeft een factory methode.
- Het Factory Method pattern heeft een abstracte factory klasse.
- Het Abstract Factory pattern gebruikt niet per se een abstracte klasse. Het kan ook een interface zijn.