Samenvatting Ontwerpen III

TIN 2 - HoGent

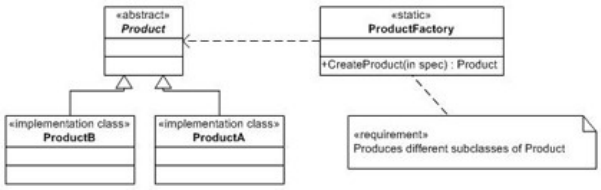
Lorenz Verschingel

22 april 2015

1 Factory Pattern

We pakken de code voor de creatie op en verplaatsen deze naar een ander object dat alleen maar het maken van producten als taak zal hebben. Dit object noemen we *Factory*.

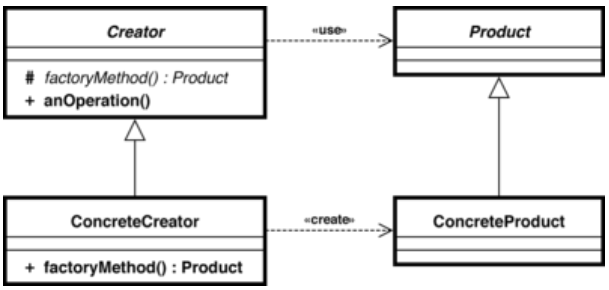
1.1 Simple Factory



Figuur 1: UML: Simple Factory

Volgens de UML in figuur 1 kan de ProductFactory producten van het type Product afleveren aan zijn cliënten.

1.2 Factory Methode



Figuur 2: UML: Factory Methode

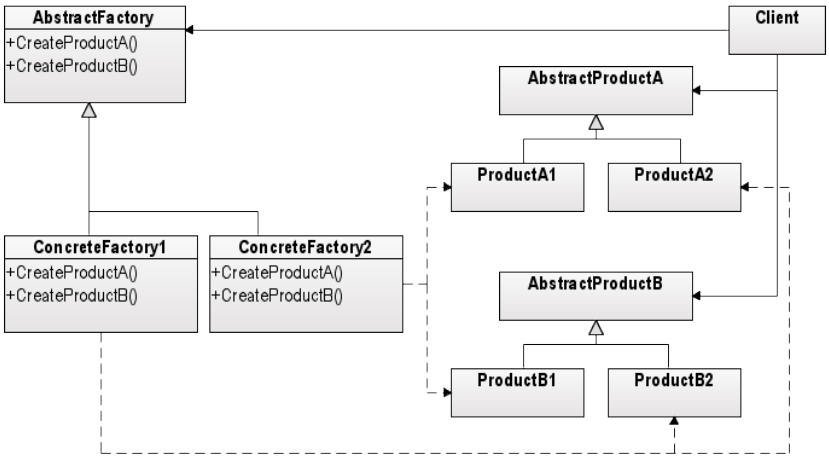
Ten opzichte van figuur 1 is er niet zoveel veranderd. In figuur 2 is de Factory klasse abstract geworden en is de create methode ook abstract. Deze wordt dan later door een concrete factory geïmplementeerd.

Het Factory Method Pattern definieert een interface voor het creëren van een object, maar laat de subklassen beslissen welke klasse er geïnstantieerd wordt. De Factory Method draagt de instanties over aan de subklassen.

1.3 Dependency Inversion-principe

Wees afhankelijk van abstracties, niet afhankelijk van concrete klassen.

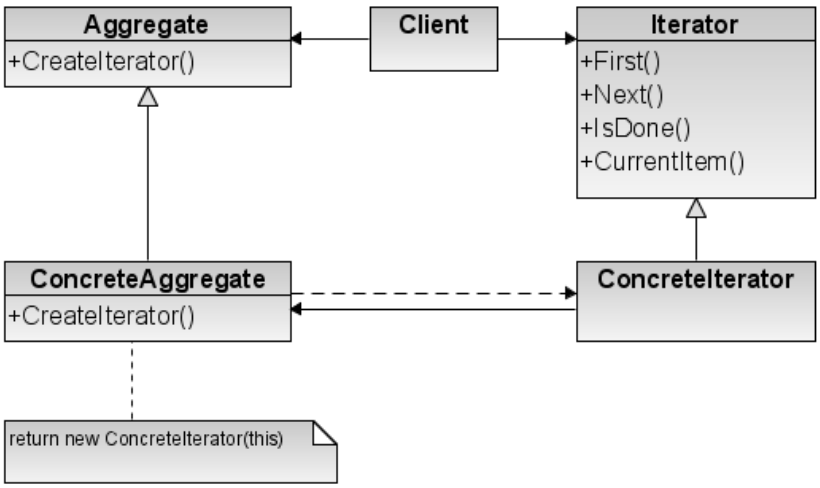
1.4 Abstract Factory Pattern



Figuur 3: UML: Abstract Factory

Het Abstract Factory Pattern levert een interface voor de vervaardiging van reeksen gerelateerde of afhankelijke objecten zonder hun concrete klassen te specificeren.

2 Iterator Pattern



Figuur 4: UML: Iterator Pattern

Het Iterator Pattern voorziet ons van een manier voor sequentiële toegang tot de elementen van een aggregaatobject zonder de onderliggende representatie weer te geven.

Figuur 4 heeft een redelijk uitgebreide verantwoordelijkheid aan de iterator. In de meeste gevallen volstaat het om een methode hasNext() en Next() in te voeren.

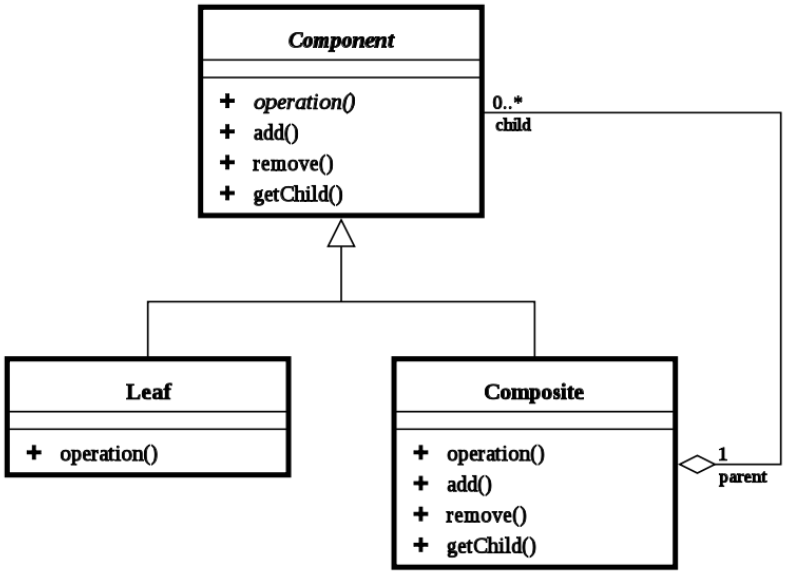
Java voorziet het iterator pattern met de klasse Iterator. Hiervoor moet men zorgen dat de iterator-klasse de klasse Iterator van Java gebruikt en dat de methode createIterator() het type Iterator<teItererenType> terug geeft.

3 Composite Pattern

Het Composite Pattern stelt je in staat om objecten in boomstructuren samen te stellen om partwhole hiërarchiën weer te geven. Composite laat clients de afzonderlijke objecten of samengestelde objecten op uniforme wijze behandelen.

Beschouw figuur 5. De klasse Component is een interface voor alle objecten in de compositie. Een leaf definiëert het gedrag voor de elementen in de compositie. Dit gebeurt via implementatie van de operaties die de klasse Composite ondersteunt. De klasse Composite definiëert het gedrag van de component met kinderen. Deze klasse moet ook alle kinderen kunnen bijhouden.

Zowel Composite als Leaf overriden enkel de methoden die zin hebben, en gebruiken de standaardimplementatie uit Component voor de methoden die niet zinnig zijn.



Figuur 5: UML: Composite Pattern

3.1 De compositie-iterator

Om een Composite-iterator te implementeren voegen we de methode createIterator() toe aan iedere component. Voor de Composite klasse geeft deze methode een iterator over zijn kinderen terug, voor de klasse Leaf een NullIterator. Als over de hele boomstructuur geïtereerd moet worden dan moet de iterator van de root in een CompositeIterator klasse verpakt worden.

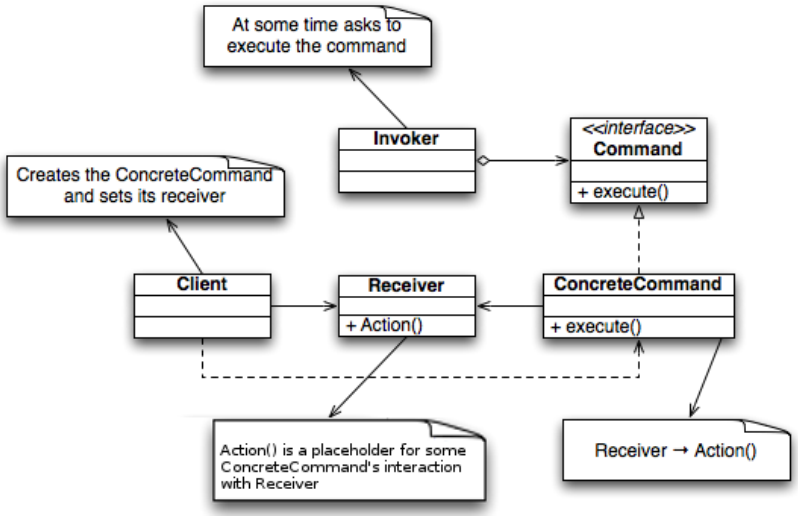
De klasse CompositeIterator implementeert de interface Iterator. Verder bevat deze klasse een stack waarop alle afzonderlijke Iterators geplaatst kunnen worden.

4 Command Pattern

Het Command Pattern schermt een aanroep af door middel van een object, waarbij je verschillende aanroepen in verschillende objecten kan opbergen, in een queue kan zetten of op schijf kan bewaren. Vaak worden ook undo-operaties ondersteund.

Het command pattern wordt gebruikt om een actie voor te stellen als een object. De commands bevatten alle info die ze nodig hebben om de actie te kunnen uitvoeren. Nieuwe commando's kunnen eenvoudig worden toegevoegd.

In figuur 6 zorgt de Client ervoor dat de Invoker het juiste commando kent. Later kan de Client dan ook aan de Invoker vragen om het commando uit te voeren.



Figuur 6: UML: Command Pattern

4.1 Macro-Command

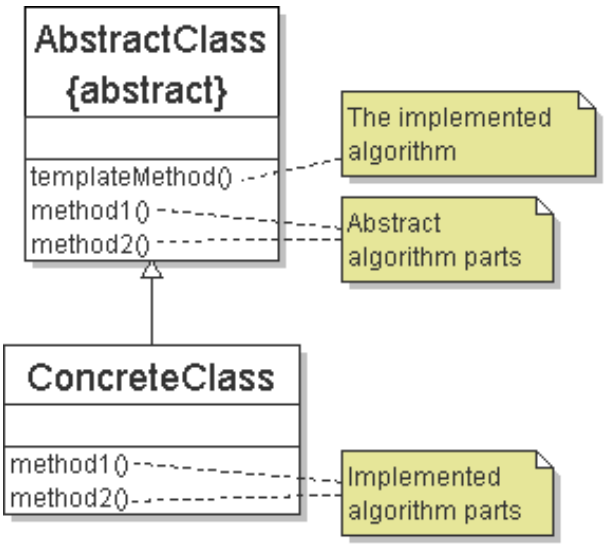
Een macro-command houdt een lijst bij met allerlei commando's. Bij het uitvoeren van het macro-commando voert deze dan alle commando's die in de lijst zitten één voor één uit.

4.2 Toepassingen

- Aanvragen in een wachtrij
- Logging aanvragen
- ASP.NET MVC: Klasse ActionResult met methode ExecuteResult().

5 Template Method Pattern

Het Template Method Pattern definieert het skelet van een algoritme in een methode, waarbij sommige stappen aan subklassen worden overgelaten. De Template method laat subklassen bepaalde stappen in een algoritme herdefiniëren zonder de structuur van het algoritme te veranderen.



Figuur 7: UML: Template Pattern

5.1 Inhaken in een Template Method

In de template method kan men een conditionele opdracht toevoegen. Deze methode heeft een (bijna) lege standaard implementatie. Deze methode noemt men dan de hook. Een subklasse kan deze methode overriden maar dit is niet verplicht.

```
public abstract class CaffeineBeverage {
  public final void prepareRecipe() {
    boilWater();
    brew();
    pourInCup();
    if (customerWantsCondiments())
      addCondiments();
  }
  protected void boilWater() {
    System.out.println("Boiling water");
  }
  protected void pourInCup() {
    System.out.println("Pouring into cup");
  }
  protected abstract void brew();
  protected abstract void addCondiments();
  protected boolean customerWantsCondiments() {
    return true;
  }}
```

Figuur 8: Code met hook

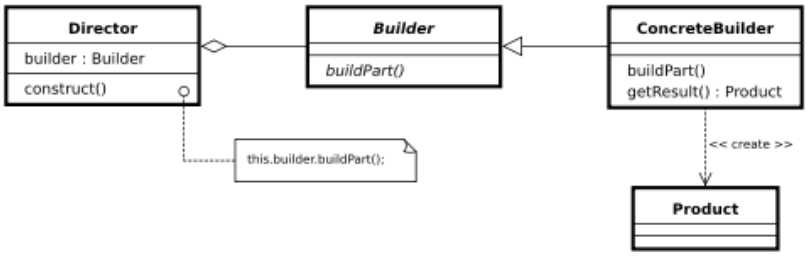
In figuur 8 is de methode customerWantCondiments() de hook. Enkel als deze methode waar is dan wordt de methode addCondiments() opgeroepen.

6 Builder Pattern

Gebruik het Builder pattern om de constructie van een product af te schermen en zorg dat je het in stappen kan construeren.

Mogelijke hints hiervoor zijn:

- Constructor met veel parameters
- Klasse met veel constructors
- Samengesteld object



Figuur 9: UML: Builder pattern

In figuur 9 is de klasse Builder een abstracte interface voor de creatie van de onderdelen van het Product object. De ConcreteBuilder bouwt de onderdelen van het complexe object en gooit deze samen door implementatie van de Builder interface. Het houdt een representatie van het object bij en biedt een interface voor het opvragen van het product. De Director klasse bouwt het complexe object gebruik makend van de interface van de Builder.

De voordelen van het Builder Pattern zijn dat de manier waarop een complex object gebouwd wordt afgeschermt is. Het geeft ook de mogelijkheid om objecten in meerdere stappen en wisselende processen te maken. (Dit in tegenstelling tot Factory Pattern.) Het verbergt de interne representatie van het product voor de client. En tot slot kunnen Productimplementaties steeds wisselen, dit is zo omdat de client alleen een abstracte interface ziet.

Een nadeel van het Builder Pattern is dat het maken van een object meer domeinkennis vereist van de client dan bij het Factory Pattern. Dit kan opgelost worden door een Director klasse toe te voegen.

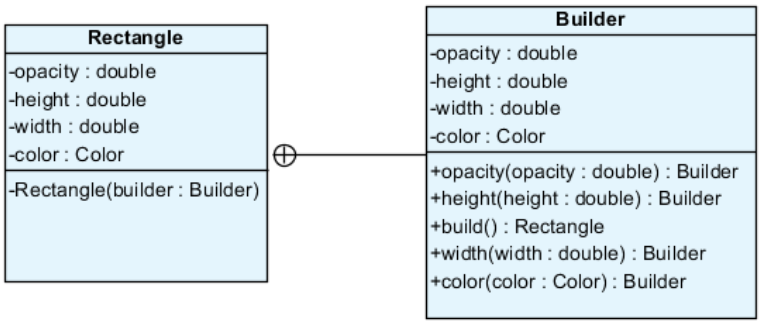
6.1 Variant

Deze variant is bedoeld voor de constructie van klasses met

- veel opties.
- veel argumenten bij constructie.

- mogelijks ook al veel default waarden.

Men kan dan de Builder klasse als een interne klasse implementeren.



Figuur 10: Voorbeeld: Variant van het Builder Pattern

Als de code bij figuur 10 geïmplementeerd wordt dan wordt bij de constructie van de rechthoek alle attributen uit de builder opgehaald. Constructie van de rechthoek gebeurt daar de static Builder aan te spreken en zo alle attributen mee te geven. Nadat alle atributen zijn gemaakt kan met met behulp van de static Builder de rechthoek construeren.

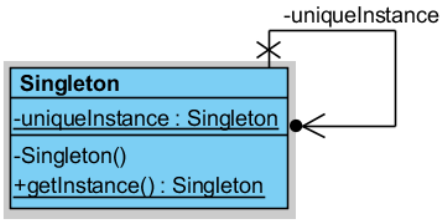
Merk ook op dat de meeste "setters"van de klasse Builder ook een Builder retourneren. Dit is dan het eigenlijke Builder-object waarop iets werd toegevoegd.

7 Singleton

Een singleton is een uniek object waarvoor er slechts één instantie bestaat. Indien er meerdere instanties van dit object zouden zijn, resulteert dit gegarandeerd in problemen. Het singleton garandeert dan ook dat er maar één en niet meer dan één instatie van een bepaalde klasse bestaat. Verder levert de klasse één globaal toegangspunt. De instatie van de klasse wordt pas aangemaakt op het moment dat dit nodig is (= Lazy Loading).

In het kort:

Het Singleton Pattern garandeert dat een klasse slechts één instantie heeft, en biedt een globaal toegangspunt ernaartoe.



Figuur 11: UML: Singleton

In figuur 11 ziet men dat het Singleton een uniek object van het type Singleton bijhoudt. Omdat dit Singleton altijd hetzelfde object moet zijn is dit attribuut static. Om dezelfde reden is de methode getInstance() ook static.

7.1 Problemen bij multithreading

Als er niet ingegrepen wordt kunnen er toch meerdere instanties van het Singleton object bestaan. Dit kan tot ernstige problemen leiden. Toch kan dit probleem relatief makkelijk opgelost worden in Java.

Een eerste oplossing is zonder lazy loading. Men zorgt dat het Singleton-object final is en er wordt ook meteen een instantie gemaakt. Hiervoor kan de code uit figuur 12a bekeken worden.

Een tweede oplossing is om de static methode getInstance synchronised te maken. Zo kan men weer gebruik maken van lazy loading, maar men dient wel te besteffen dat synchronisatie duur is. Het kan de performatie met een factor 100 reduceren. Ook is de synchronisatie overbodig nadat de code voor de eerste maal is doorlopen. De code hiervoor kan in figuur 12b bezichtigd worden.

```
public class Singleton {

  private static final Singleton uniqueInstance
      = new Singleton();

  private Singleton() {
  }

  public static Singleton getInstance() {
    return uniqueInstance;
  }

}
```

(a) Singleton met Eager Loading

```
public class Singleton {

  private static Singleton uniqueInstance;

  private Singleton() {
  }

  public static synchronized Singleton getInstance() {
    if (uniqueInstance == null) {
      uniqueInstance = new Singleton();
    }
    return uniqueInstance;
  }
}
```

(b) Singleton met Lazy Loading

Figuur 12: Code: multithreading safe

8 Proxy Pattern

Het proxy pattern wordt gebruikt om de objecttoegang te controlleren.

8.1 De rol van de remote proxy

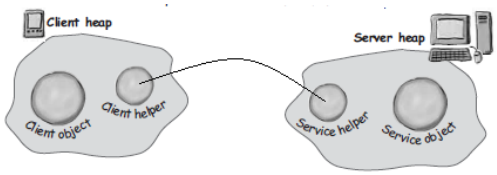
De remote proxy funcgeert als een lokale vertegenwoordiger van een remote object.

Het lijkt alsof een client remote-aanroepen pleegt. In werkelijkheid worden methoden aangeroepen van een proxy object in de local heap, die alle low-level details van de netwerkcommuncatie afhandelt.

8.2 Remote Method Invocation

Remote Method Invocation (RMI) laat toe om objecten te vinden in een remote JVM en hun methodes aan te roepen.

8.2.1 Werking van RMI



Figuur 13: Werking RMI

1. Het client object roept de methode aan van de client helper.
2. De client helper verpakt de info over de aanroep en stuurt deze over het netwerk naar de service helper.
3. De service helper pakt de info uit, ontdekt welke methode moet worden opgeroepen en roept de echte methode voor het echte service object aan.
4. Het service object voert de methode uit en retourneert het resultaat naar de service helper.
5. De service helper verpakt het resultaat en verstuurt dit over het netwerk naar de client helper.
6. De client helper pakt het resultaat uit en lever het aan het client object.

Al het bevenstaande was transparant voor het client object.

8.2.2 Java RMI

De client helper (proxy in de client heap) wordt hierbij de RMI stub genoemd, de service helper (proxy in de server heap) wordt het RMI skeleton genoemd.

De RMI stub gaat fungeren als een proxy. De RMI-skeleton hoeft in de nieuwere Java versies niet meer expliciet gebruikt te worden. Java RMI zorgt er wel nog steeds voor dat iets het skeletongedrag afhandelt aan de serverkant.

8.2.3 De remote service maken

1. Remote interface maken
    - Definieert de remote methoden die de client kan aanroepen
    - Breidt de klasse Java.rmi.Remote uit.
    - Declareer dat alle methoden een RemoteException kunnen gooien
    - Zorg ervoor dat alle argumenten en retournwaarden primitief of serialiseerbaar zijn.
    - Objecten die niet geserialisser moeten worden kan men aangeven met het keyword "transient".
2. Remote implementatie maken
    - Deze klasse doet het echte werk. Dit is het object waarvan de client methodes wil aanroepen.
    - Implementeer de remote interface.
    - Breid UnicastRemoteObject uit (zo kan het van buitenaf worden aangesproken.
    - Schrijf een constructor die een RemoteException kan gooien.
3. Stub en skeleton worden dynamisch aangemaakt door JVM (sinds Java 1.5). Dit gebeurt automatisch en hoeft niet noodzakelijk manueel gedaan te worden.
4. RMI registry starten (=name service voor remote objecten)
    - Registry registry = LocateRegistry.createRegistry(1099);
5. Het remote object bekend maken bij de name service
    - Hiervoor dient men het remote object instantiëren en in het RMI registery te plaatsen (via registry.rebind()).
6. Run

8.2.4 De remote service ophalen

Via de RMI Registry

1. Client zoekt RMI Registery.
    - Registry registry = LocateRegistry.getRegistry("127.0.0.1", 1099);
2. RMI Registry retourneert het stubobject.
    - registry.lookup("ProxyName");
3. Client roept methode van stub aan, alsof de stub de echte service is.