Design Patterns

Inhoud

- Programming to an interface
- Inversion Of Control
  - Dependency Injection
- Factory
- Singleton
- Model View Controller

Programming to an interface

= Code moet communiceren op basis van een gedefinieerde set van functionaliteiten in plaats van een expliciet gedefinieerd objecttype.

→ Deze techniek wordt vaak gebruikt als we bezig zijn over ‘Inversion of control’.

Deze 2 klassen hebben niets met elkaar gemeen. Maar stel dat we een spel aan het maken zijn waarbij er irritante mensen/dieren/objecten u het leven zuur maken.

→ Key: Zowel de vlieg als de Telemarketeer zouden één gemeenschappelijk gedrag moeten interpreteren.

```java
package be.intec.model;

public class Fly extends Insect {

    public void flyAroundYourHead() {

    }

    public void landOnThings() {

    }

}
```

```java
package be.intec.model;

public class Telemarketer extends Person {

    public void callDuringDinner() {

    }

    public void continueTalkingWhenYouSayNo() {

    }

}
```

```java
package be.intec.interfaces;

public interface IPest {
    void beAnnoying();
}

```

```java
package be.intec.model;

import be.intec.interfaces.IPest;

public class Fly extends Insect implements IPest {

    public void flyAroundYourHead() {

    }

    public void landOnThings() {

    }

    @Override
    public void beAnnoying() {
        flyAroundYourHead();
        landOnThings();
    }
}

```

```java
package be.intec.model;

import be.intec.interfaces.IPest;

public class Telemarketer extends Person implements IPest {

    public void callDuringDinner() {

    }

    public void continueTalkingWhenYouSayNo() {

    }

    @Override
    public void beAnnoying() {
        callDuringDinner();
        continueTalkingWhenYouSayNo();
    }
}
```

Beide klassen implementeren allebei op hun eigen manier de methode om irritant te zijn.

Als de personen aan het dineren zijn dan komen de instanties van IPest (Vlieg en Telemarketeer) irritant doen.

```java
package be.intec.model;

import be.intec.interfaces.IPest;

public class DiningRoom {
    private IPest[] pests;

    public DiningRoom(Person[] diningPeople, IPest[] pests) {
        this.pests = pests;
    }

    void ServeDinner() {
        // Als de mensen aan het eten zijn:
        for (IPest p : pests) {
            p.beAnnoying();
        }
    }
}
```

Inversion Of Control

= Een programmeer techniek, waarbij objecten worden gekoppeld tijdens de uitvoering (runtime).

→ We geven de controle over.

Dependency Injection

= een vorm om van ‘inversion of control’, die het mogelijk maakt klassen losjes te koppelen (loose coupling). Dit wil zeggen dat ze data kunnen uitwisselen zonder dat deze relatie hard (in de broncode) vastgelegd is; althans niet door de programmeurs van die (beide) klassen.

Voorbeeld

Je hebt een lamp en een (snoer)schakelaar. Deze objecten werken allebei los van elkaar. De lamp kan gebruik maken van verschillende schakelaars, zolang deze maar worden voorzien van elektriciteit. **= LOOSE COUPLING**

Daarnaast kan je ook een staanlamp kopen die een vaste schakelaar heeft. Hier is het niet mogelijk om een andere schakelaar te gebruik om de lamp te doen branden **= TIGHT COUPLING**

Constructor Injection

→Referentie naar het object gebeurt in de constructor.

```
public class clsCustomer
{

private IAddress _address;

public clsCustomer(IAddress obj)
{
        _address = obj;
}
.........
.........
}
```

Object injected through constructor

Setter Injection

→Referentie naar het object gebeurt in de setter methode.

```
public class clsCustomer
{

private IAddress _address;

public IAddress Address
{
    set
    {
        _address = value;
    }
}
.........
.........
}
```

Using Set Propertise

Interface Injection

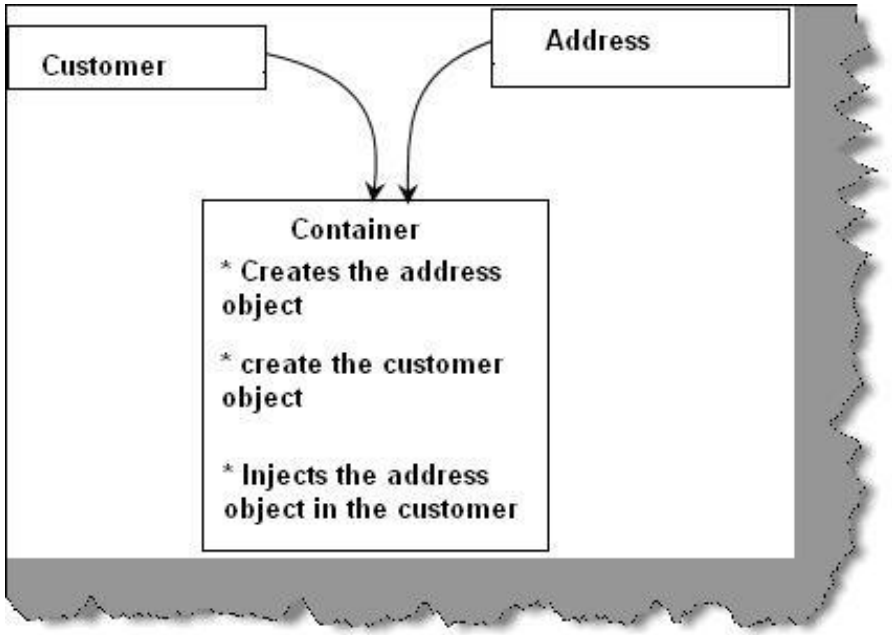
→Referentie naar het object gebeurt met behulp van een Interface.

```
interface IAddressDI
{
  void setAddress(IAddress obj);
}
```

```
public class clsCustomer : IAddressDI
{

private IAddress _address;

public void setAddress(IAddress obj)
{
  _address = obj;

}
.........
.........
}
```

DI Container

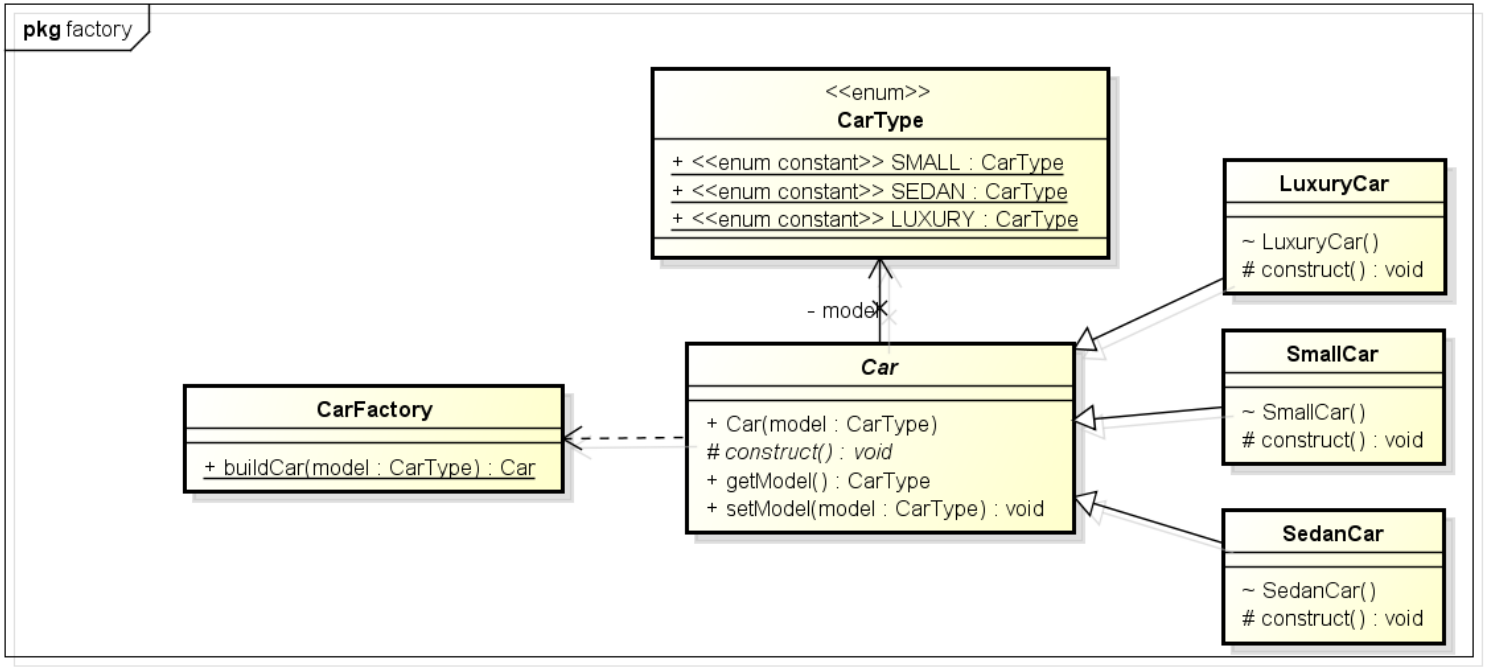
Dependency Injection Container maken die verantwoordelijk is voor het beheer, instantiatie en configuratie van het object.



Factory Pattern

= een manier om objecten te instantiëren zonder exact vast hoeven te leggen van welke klasse deze objecten zullen zijn.

Factory Pattern



Factory dient om een instantie van een auto te maken, alleen als het bekend is welk type er gemaakt moet worden.

Singleton Pattern

= Je kan maar één instantie aanmaken van de klasse.

```
public class SingleObject {

   //create an object of SingleObject
   private static SingleObject instance = new SingleObject();

   //make the constructor private so that this class cannot be
   //instantiated
   private SingleObject(){}

   //Get the only object available
   public static SingleObject getInstance(){
      return instance;
   }

   public void showMessage(){
      System.out.println("Hello World!");
   }
}
```

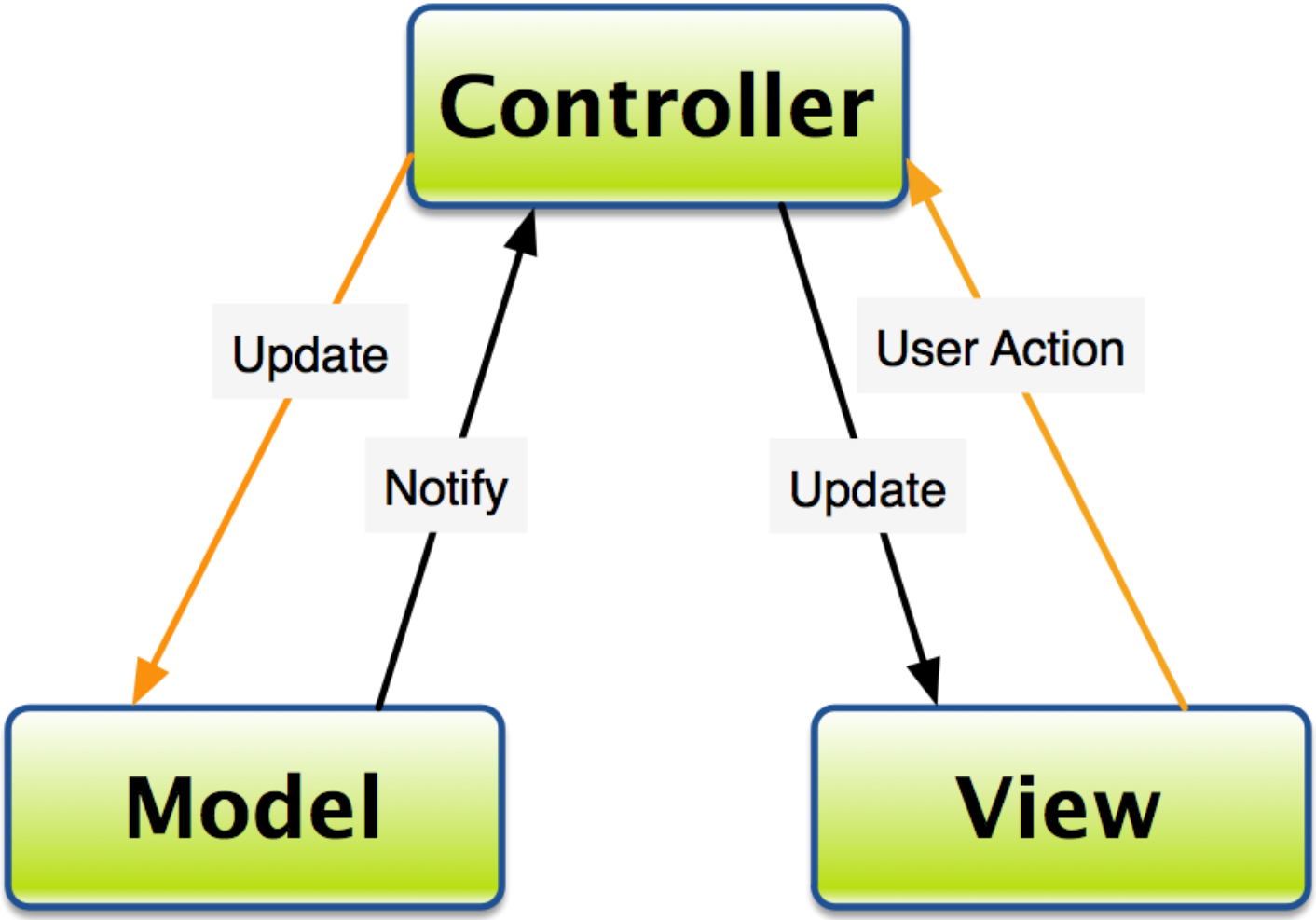
Model View Controller

Model: Object + logica om controller te updaten als data verandert.

View: Visualisatie vd gegevens van het model.

Controller: Werkt zowel op model als op view. Controleert het model en wanneer gegevens veranderen wordt de view geüpdate.

Model View Controller



PRO

- Hergebruik van code
- Code is beter leesbaar/ duidelijker
- Faster development
- Code is beter onderhoudbaar

CONTRA

- Neemt meer geheugen in beslag