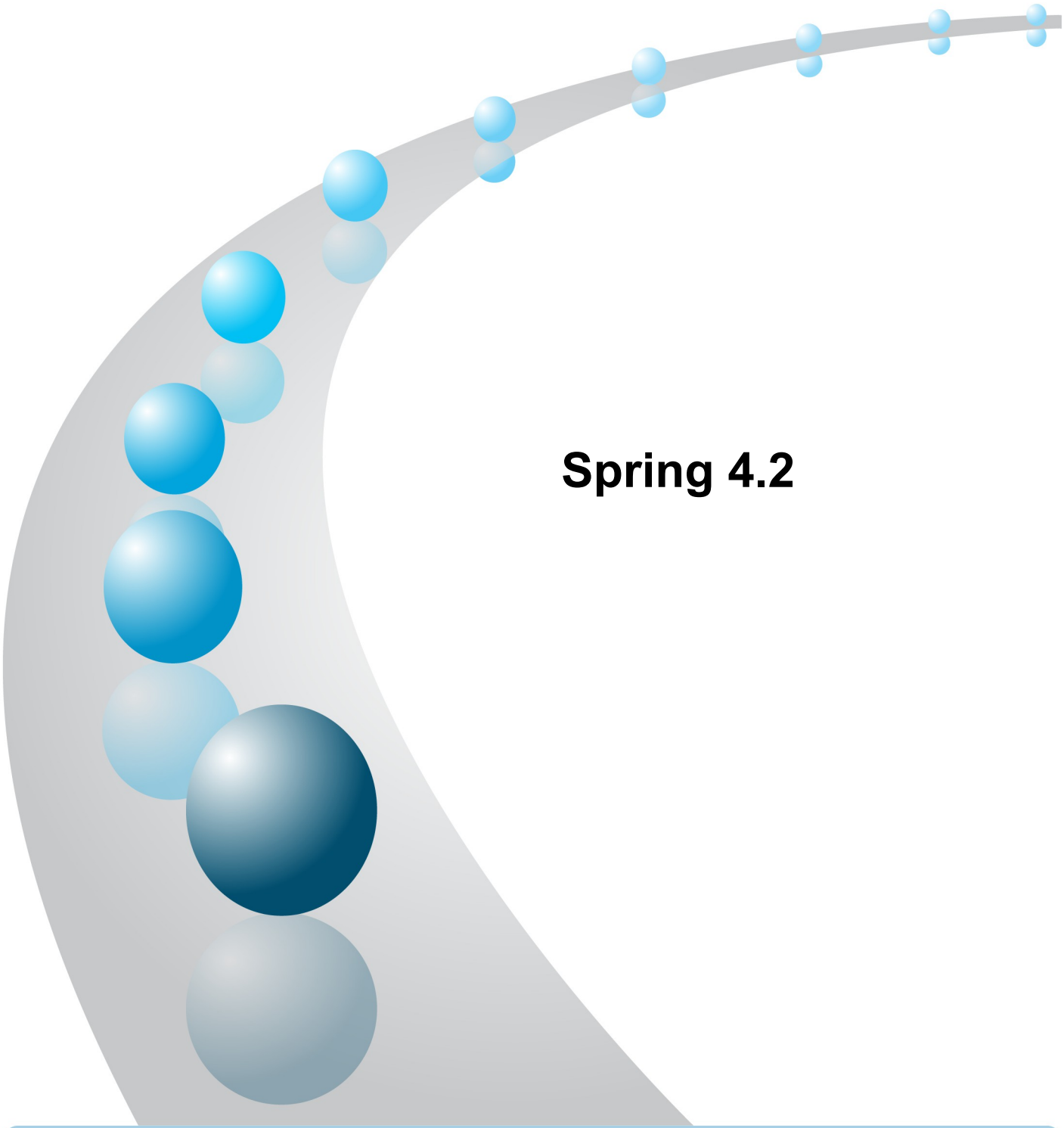
Noël Vaes

Java Trainer & Consultant



Spring 4.2

Roode Roosstraat 5
3500 Hasselt
België

+32 474 38 23 94
noel@noelvaes.eu
www.noelvaes.eu

Correspondentie inzake overnemen of reproductie kunt u richten aan:

Noël Vaes
Roode Roosstraat 5
3500 Hasselt
België

Tel: +32 474 38 23 94

noel@noelvaes.eu
www.noelvaes.eu

24/09/2016

Inhoudsopgave

Hoofdstuk 1: Core Spring

1.1 Inleiding

Spring is een *open source framework* voor de ontwikkeling van Java-toepassingen met behulp van herbruikbare en configureerbare componenten in de vorm van *plain old Java objects (POJO's).* Wat dat precies betekent zullen we in de loop van deze cursus stap voor stap duidelijk maken. In feite kunnen we allerlei soorten toepassingen met *Spring* ontwikkelen maar vaak zijn het *multitier* gedistribueerde applicaties.

Deze cursus is gericht op het ontwikkelen van dergelijke *multitier* gedistribueerde applicaties met *Spring*. Daarom zullen we eerst wat meer uitleg geven over de architectuur van dit soort applicaties.

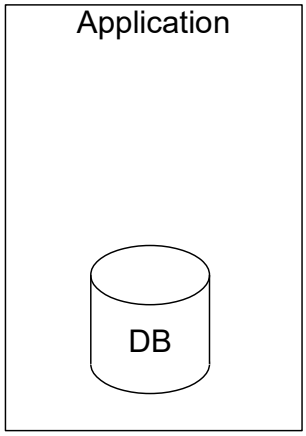
1.1.1 *Multitier* gedistribueerde applicaties

Multitier gedistribueerde applicaties zijn toepassingen waarbij de functionaliteit verspreid ligt over meerdere systemen die d.m.v. een netwerk met elkaar verbonden zijn. De software wordt onderverdeeld in verschillende fysieke lagen met elk hun eigen verantwoordelijkheid.

Om de noodzaak van dat soort applicaties aan te tonen, geven we even een overzicht van de verschillende soorten applicaties.

1.1.1.1 *One tier*-applicaties

De meest eenvoudige applicaties zijn de *one tier*-applicaties. Heel de functionaliteit is vervat in de applicatie en deze kan bijgevolg volledig zelfstandig uitgevoerd worden.



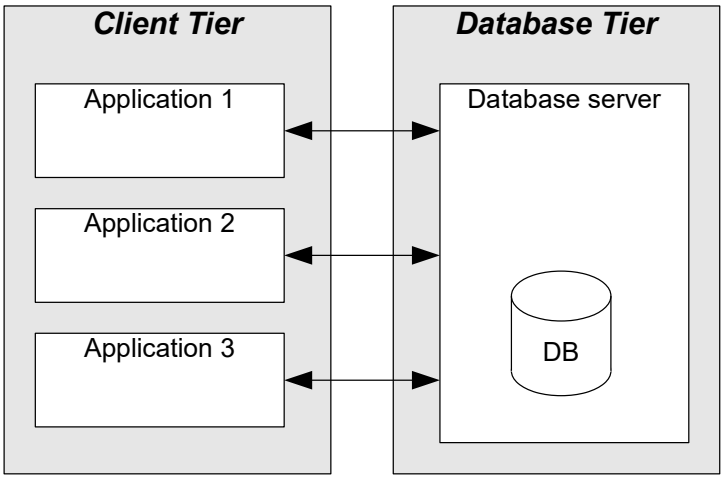
Afbeelding 1: One tier-applicatie

Vaak wordt er in dat soort applicaties gebruik gemaakt van een database. Deze is dan vervat in de applicatie zelf. Men spreekt dan van een *embedded database*. Dit soort applicaties is goed indien er geen informatie gedeeld moet worden met andere applicaties, eventueel met andere instanties van dezelfde applicatie. Iedere applicatie staat volledig op zichzelf en is niet verbonden met andere applicaties. Alle functionaliteit wordt uitgevoerd op het lokale systeem.

1.1.1.2 *Two tier*-applicaties

Meestal is het echter nodig dat een applicatie informatie deelt met andere applicaties, eventueel met andere instanties van dezelfde applicatie. Hierbij wordt het opslaan van de

gegevens afgezonderd uit de applicatie en toevertrouwd aan een tweede applicatie die voor meerdere toepassingen toegankelijk is. Indien de data bewaard wordt in een database, wordt gebruik gemaakt van een databaseserver.



Afbeelding 2: Two tier-applicatie

De software wordt daarbij verspreid over twee *tiers*. Vooreerst is er de *client tier* die de applicatie bevat waarmee de gebruiker werkt. Voorts is er de *database tier* die de databaseserver bevat. Deze is doorgaans geplaatst op een andere machine in het netwerk. De communicatie tussen de applicatie en de database server verloopt dan via het netwerk (meestal op basis van het TCP/IP-protocol).
De gegevens die bewaard worden door de *database server* zijn toegankelijk vanuit verschillende applicaties. Dit maakt het mogelijk dat deze applicaties hun gegevens delen.

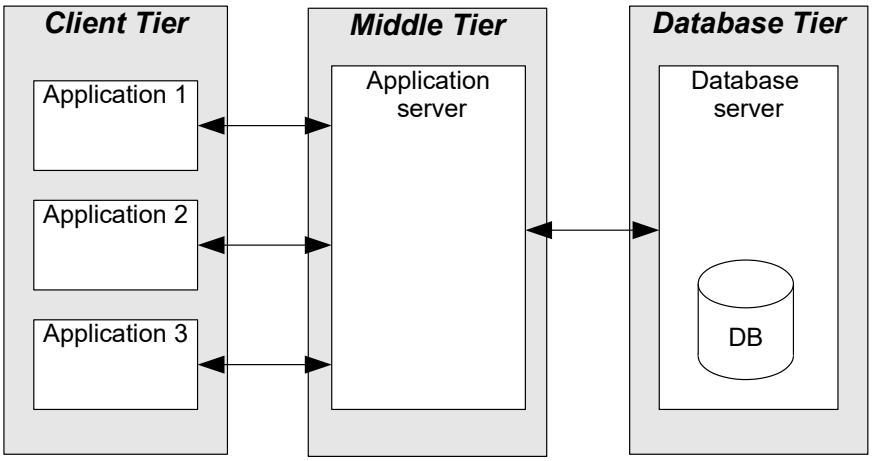
1.1.1.3 *Three tier-applicaties*

Bij *two tier*-applicaties bevindt zich heel de applicatielogica in de *client tier*. Tevens bevat deze *tier* ook alles om de gebruikersinterface te presenteren. We noemen dit ook wel de presentatielogica.

Het is echter mogelijk dat eenzelfde applicatie verschillende soorten gebruikersinterfaces heeft. Denk maar aan een applicatie met zowel een webinterface als een *Swing*-interface.

In het *two tier*-model moeten we voor iedere gebruikersinterface een afzonderlijke applicatie maken met de eigen presentatielogica. Vermits ook de applicatielogica vervat is in de applicatie moeten we deze daarin telkens opnieuw voorzien.

Het zou echter beter zijn de applicatielogica af te zonderen van de presentatielogica. Dit komt de herbruikbaarheid van de softwarecomponenten ten goede. Dit resulteert in een *three tier*-applicatie.



Afbeelding 3: Three tier-applicatie

De middelste *tier* bevat de *application server* die de applicatielogica bevat. Hiermee bedoelen we uiteindelijk alle functionaliteit die niet onmiddellijk gerelateerd is aan de presentatie van de applicatie aan de gebruiker. We spreken in het algemeen van *middleware*; dit is software die zich in het midden bevindt.

Deze *application server* bevindt zich doorgaans ook op een afzonderlijke machine in het netwerk. Verschillende applicaties kunnen simultaan gebruik maken van de *application server.* De eindapplicaties moeten nu enkel nog zorgen voor de aangepaste presentatie van de applicatie naar de gebruiker toe. Zo kunnen verschillende applicaties met totaal verschillende gebruikersinterfaces toch samen gebruik maken van dezelfde applicatielogica. Vermits de applicaties enkel nog de presentatielogica bevatten, zijn ze dus vaak erg afgeslankt. Men spreekt in dat geval ook wel van *thin clients*.

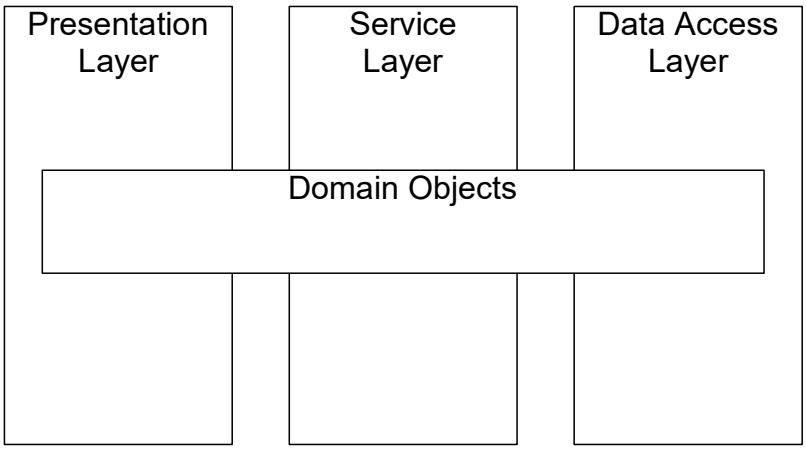
1.1.2 *Layers* of logische lagen

Tiers zijn de fysieke lagen waarover de applicatie verspreid is. Daarnaast spreekt men vaak ook over de logische lagen of *layers.* Dit zijn de lagen waarin de code georganiseerd is. Deze logische lagen zijn niet noodzakelijk gebonden aan een fysieke laag. Soms bevinden meerdere logische lagen zich op eenzelfde fysieke laag en in andere gevallen kan een logische laag verspreid zijn over meerdere fysieke lagen.

Doorgaans onderscheidt men volgende logische lagen in een applicatie:

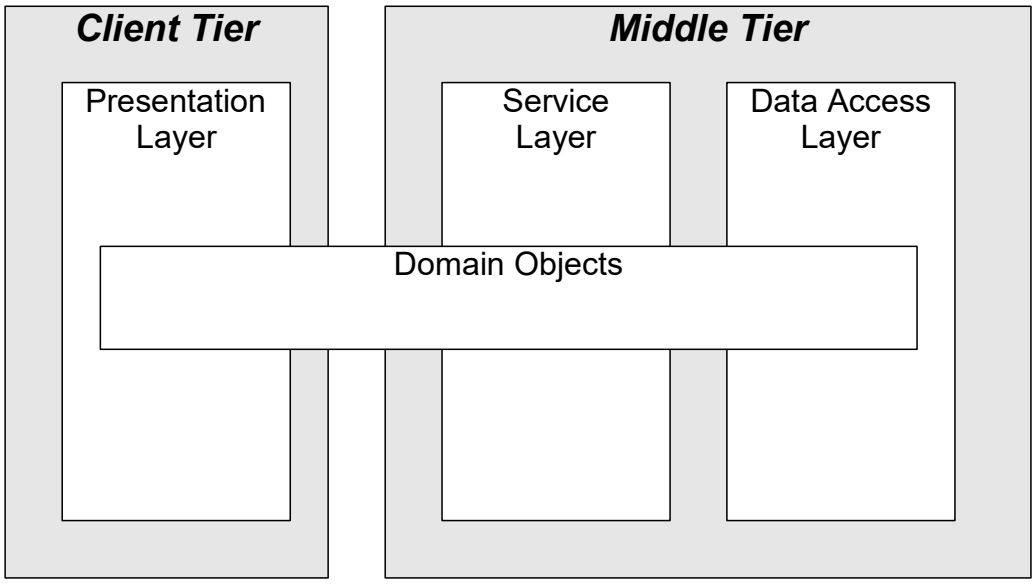
1. ***Data Access Layer***: Deze laag is verantwoordelijk voor de communicatie met de databank. De toegang tot de databank is hier gecentraliseerd en afgescheiden van de rest.
2. ***Service Layer***: In deze laag wordt de *business logic* uitgevoerd. Deze bestaat uit allerlei diensten (*services*) ten behoeve van o.a. de *Presentation Layer*.
3. ***Presentation Layer***: Deze laag voorziet de presentatie van de applicatie naar de eindgebruiker toe. Tenminste in het geval er een grafische gebruikersinterface nodig is. Bij een B2B (*business to business)* toepassing is dit niet noodzakelijk het geval.
4. ***Domain Objects:*** In de gehele applicatie zijn er meestal data-objecten nodig. Deze worden in de *Data Access Layer* gesynchroniseerd met de databank. In de *Service Layer* worden deze objecten gemanipuleerd en in de *Presentation Layer* worden ze gebruikt om gegevens te tonen en nieuwe invoer van de gebruiker over te brengen naar de *Service Layer*. Deze objecten worden dus gebruikt in de drie andere lagen en om die reden wordt in de tekening deze blok dwars weergegeven.

In onderstaande afbeelding geven we de verschillende lagen weer:



Afbeelding 4: De logische lagen in een applicatie.

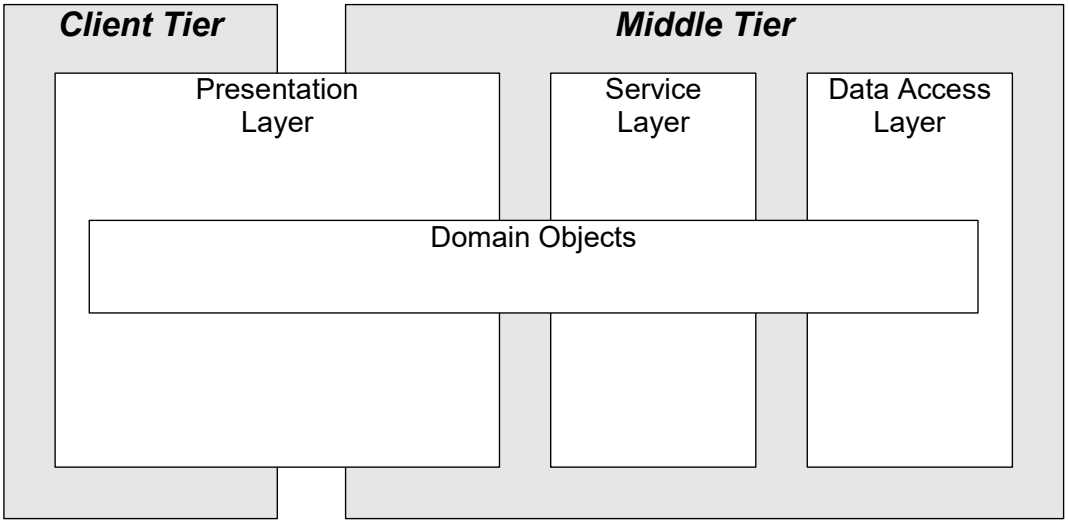
Deze lagen kunnen zich nu op verschillende fysieke lagen bevinden. In onderstaande afbeelding bevindt de *Presentation Layer* zich volledig op de *client tier*. Dit zou het geval kunnen zijn indien de grafische gebruikersinterface ontworpen is als een toepassing in Swing of Java-FX.



Afbeelding 5: Presentation Layer op de client tier

Ook toepassingen op basis van *JavaScript frameworks* zoals bijvoorbeeld *JQuery* of *AngularJS* komen overeen met deze architectuur. De presentatielaag is daarbij volledig geschreven met behulp van HTML, CSS en *JavaScript.*

Bij een klassieke dynamische webtoepassing daarentegen is de *Presentation Layer* meestal gespreid over de *client tier* en de *middle tier*. De HTML-pagina's worden dynamisch aangemaakt op de webserver en getoond in de browser.



Afbeelding 6: Presentation Layer op de client tier en middle tier

1.1.3 Java Enterprise Edition

Om dergelijke *multitier* gedistribueerde applicaties met meerdere logische lagen te maken kan men best beroep doen op een bestaande software-infrastructuur die een aantal functionaliteiten kant en klaar aanbiedt.

Om ervoor te zorgen dat de verschillende aanbieders van dergelijke infrastructuur hun diensten op dezelfde manier aanbieden, is in de Java-wereld de *Java Enterprise Edition (JEE)* ontwikkeld. Dit is een uitbreiding op de *Java Standard Edition (JSE)* en bevat hoofdzakelijk specificaties waaraan de aanbieders moeten voldoen. Technisch gaat het hierbij o.a. om allerlei interfaces waarvoor men een implementatie moet voorzien.

JEE bevat de nodige technologieën om een *multitier* gedistribueerde applicatie te maken waarbij de ontwikkelaar zich kan concentreren op zijn specifieke logica en visuele voorstelling en waarbij de algemene dingen worden afgehandeld door een platform dat de JEE-specificaties implementeert.

Het JEE-verhaal met vooral het onderdeel EJB (*Enterprise JavaBeans)* heeft echter een bewogen geschiedenis. De complexiteit van EJB 2 was erg hoog en dat maakt dat dit onderdeel door de gemeenschap van ontwikkelaars nauwelijks werd aanvaard. Intussen is men met EJB 3 een nieuwe richting ingeslagen die terug beter aanslaat. Intussen hadden veel ontwikkelaars wel reeds afgehaakt en waren op zoek gegaan naar betere alternatieven: o.a. *Spring* en *Hibernate*.

1.1.4 Spring

Spring is ontstaan als alternatief voor de complexe EJB's (vooral EJB 2.x). Men is hierbij teruggegaan naar de eenvoud van gewone Java-objecten (*plain old Java objects*) die in uiteenlopende omstandigheden gebruikt kunnen worden; dit kan zowel binnen een *standalone client* applicatie, binnen een web-applicatie of binnen eender welke laag van een *multitier*-applicatie.

1.1.4.1 Kenmerken van Spring

We sommen even een aantal belangrijke kenmerken van *Spring* op:

1. *Spring* is ***lightweight*****:** het gebruik van *Spring* vergt geen lijvige JAR-bestanden en ook de *overhead* tijdens de uitvoering is beperkt. Er is dus geen zware

applicatieserver nodig die veel systeembronnen nodig heeft, zoals dat wel het geval is bij EJB.

2. *Spring* is ***nonintrusive*****:** de componenten zijn niet afhankelijk van *Spring* of deze afhankelijkheid wordt zo klein mogelijk gehouden zodat dergelijke objecten ook elders gebruikt kunnen worden.
3. ***Dependency injection*****:** *Spring* promoot de losse koppeling tussen componenten waarbij de afhankelijkheden van buitenaf geïnjecteerd worden in plaats van intern opgezocht of gecreëerd worden. We zullen hier later nog uitvoerig op terugkomen.
4. ***Aspect oriented*****:** *Spring* maakt gebruik van *aspect oriented* technieken om de objecten te voorzien van een functionaliteit die niet tot hun kerntaak behoort, de zogenaamde *cross cutting concerns*. Ook dit zullen we uitvoerig behandelen.
5. ***Container*****:** *Spring* is een soort mini-container doordat hij instaat voor de levenscyclus van Java-objecten.
6. ***Framework*****:** *Spring* is tevens een *framework* dat een aantal belangrijke taken voor zijn rekening kan nemen zodat de ontwikkelaar zich enkel hoeft bezig te houden met de specifieke logica van zijn toepassing.

1.1.4.2 Spring modules

Spring is opgebouwd rond een eenvoudige kern (*Spring Core*) die verder uitgebreid kan worden met allerlei modules. In deze cursus beginnen we met *Spring Core.* Deze kern kan nadien uitgebreid worden met allerlei modules die ons extra mogelijkheden bieden. Het aantal extra modules is zeer groot en in deze cursus maken we een selectie van enkele modules die het meest gangbaar zijn.

Zo breiden we in het hoofdstuk *Spring Enterprise* de kern uit met extra modules voor de integratie van JPA/Hibernate en transactiebeheer. Tevens voegen we modules toe om onze software vanop afstand ter beschikking te stellen via verschillende protocollen. Dit noemt men ook wel *remoting*.

In het hoofdstuk *Web Spring* voegen we dan weer andere modules toe voor het maken van een webapplicatie gebaseerd op de MVC-architectuur.

Dit alles zal duidelijk worden naarmate we voortschrijden in de cursus. Laten we alvast beginnen met de installatie van het *Spring framework*.

1.2 Installatie van Spring

Alle informatie over het *Spring framework* is te vinden op de volgende locatie: http://spring.io.

De nodige JAR-bestanden kan men eventueel afhalen op de volgende locatie: http://repo.spring.io/release/org/springframework/spring/.

Spring is geen applicatieserver met een specifieke installatie- en opstartprocedure. *Spring* bestaat gewoon uit een aantal JAR-bestanden die opgenomen moeten worden in het *classpath* van de applicatie. Zelf maakt *Spring* ook gebruik van andere *open source* projecten. De JAR-bestanden van deze afhankelijke projecten zijn evenwel niet opgenomen de *Spring*-distributie en moeten daarom ook afzonderlijk gedownload worden en toegevoegd worden aan het *classpath.*

Gezien de veelheid van afhankelijkheden en de complexiteit van de configuratie is het daarom aangewezen gebruik te maken van hulpsystemen zoals *Maven, Gradle* of *Ivy.* Deze zijn in staat de afhankelijkheden automatisch van het internet te plukken en te integreren in het project. In deze cursus maken we gebruik van *Maven*.

In de POM moeten we volgende *dependency* toevoegen:

```
<dependencies>
   <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-context</artifactId>
      <version>4.2.6.RELEASE</version>
    </dependency>
</dependencies>
```

Later zullen we nog extra afhankelijkheden toevoegen voor bijkomende modules. Om ervoor te zorgen dat alle versies consequent hetzelfde zijn en kunnen samenwerken, kunnen we gebruik maken van een *Bill of Materials (BOM)*. Deze wordt als volgt in de POM geconfigureerd:

```
<dependencyManagement>
   <dependencies>
      <dependency>
         <groupId>org.springframework</groupId>
         <artifactId>spring-framework-bom</artifactId>
         <version>4.2.6.RELEASE</version>
         <type>pom</type>
         <scope>import</scope>
      </dependency>
   </dependencies>
</dependencyManagement>
<dependencies>
   <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-context</artifactId>
   </dependency>
</dependencies>
```

Het is dan niet meer nodig het versienummer voor iedere afhankelijkheid op te geven. Dit wordt overgenomen uit de BOM.

1.3 *Dependency Injection*

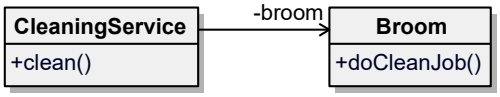
Zoals we in onze inleiding al vertelden is *Spring* een *framework* om toepassingen te maken met herbruikbare en configureerbare componenten op basis van *plain old Java objects (POJO's)*. Met *Spring* kunnen we dergelijke objecten, of ook wel *beans* genoemd, aaneenrijgen tot een werkend geheel. Dit aaneenrijgen doen we door middel van *dependency injection*.

Voor we beginnen met het gebruik van *Spring* willen we daarom even stilstaan bij deze belangrijke techniek van *dependency injection* en het concept van *inversion of control*. Beide vormen immers de grondslag van *Spring*.

1.3.1 Inversion Of Control

Spring wordt vaak omschreven als een container voor *Inversion Of Control*. Dit principe komt erop neer dat Java-objecten (*beans*) hun afhankelijkheid van andere objecten van buitenaf krijgen toebedeeld in plaats dat ze zelf intern deze afhankelijkheden moeten creëren.

We zullen dit onmiddellijk illustreren met een dagdagelijks voorbeeld. Stel dat we in ons huishouden beroep willen doen op een poetsdienst die gebruik maakt van een borstel om het huis schoon te houden. De poetsdienst wordt voorgesteld door een object van de klasse `CleaningService` die op zijn beurt gebruik maakt van een object van de klasse `Broom`.



Afbeelding 7: Klassendiagram CleaningService - Broom

De code ziet er dan als volgt uit:

```
public class Broom {
   public void doCleanJob() {
      System.out.println("Scrub scrub");
   }
}
```

```
public class CleaningService {
   private Broom broom = new Broom();

   public void clean() {
      System.out.println("Cleaning the house");
      broom.doCleanJob();
   }
}
```

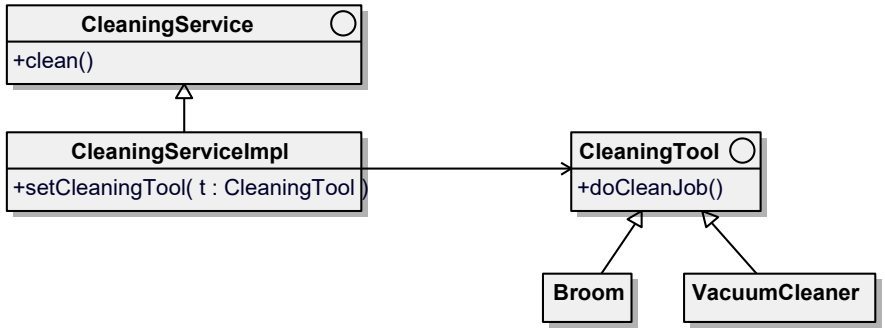
De hoofdapplicatie zou als volgt kunnen zijn:

```
public class HouseApp {
   public static void main(String[] args) {
      CleaningService cleaningService = new CleaningService();
      cleaningService.clean();
   }
}
```

In dit voorbeeld hoeven we enkel een nieuw object van de klasse `CleaningService` te instantiëren en vervolgens te vragen zijn job te doen. De poetsdienst maakt voor zijn job gebruik van een bezem waar hij zelf voor zorgt; het object van de klasse `Broom` wordt door de `CleaningService` zelf geïnstantieerd.

We zouden hier kunnen zeggen dat de controle over het poetsgerei bij de poetsdienst ligt. Het gebruik van een borstel is als het ware ingebakken in de poetsdienst. In het huishouden wordt er echter toch iets meer flexibiliteit verwacht en zou een poetsdienst ook overweg moeten kunnen met ander poetsgerei zoals bijvoorbeeld een stofzuiger.

Een beter concept is daarom een poetsdienst die je eender welk poetsgerei kan geven en die vervolgens zijn werk daarmee doet. Dit wordt weergegeven in onderstaand klassendiagram:



Afbeelding 8: Klassendiagram met Inversion Of Control

We introduceren hier de interface `CleaningTool` waarin we het gedrag van poetsgerei beschrijven. Tevens voorzien we in de klassen `Broom` en `VacuumCleaner` die beide deze interface implementeren.

Ook van de `CleaningService` maken we een interface met implementerende klasse `CleaningServiceImpl`.

De `CleaningService` maakt nu gebruik van een object dat de interface `CleaningTool` implementeert om zijn job te kunnen doen. Welk soort poetsgerei gebruikt wordt, is niet meer van belang, zolang het maar aan de voorwaarden voldoet van poetsgerei; de interface dus. Zo zie je hoe polymorfisme ook in het huishouden heel wat flexibiliteit creëert.

Dergelijke poetsdienst kan je eender welk poetsgerei in de handen duwen en die gaat daar vervolgens mee aan de slag. Het is dus niet meer de poetsdienst die beslist dat hij alleen maar met een borstel kan werken. De controle over het gereedschap ligt dus niet meer bij de poetsdienst, maar buiten hem; doorgaans de man of vrouw des huizes. Dit is dus *inversion of control*. Door dit principe is de poetsdienst dus veel flexibeler en kan voor meerdere taken ingezet worden. In software-termen kunnen we zeggen dat de klasse `CleaningService` beter herbruikbaar is; en dit is een belangrijk principe bij het ontwikkelen van software.

In feite hebben we hier voldaan aan het principe '*high cohesion and loose coupling*". *High cohesion* wil zeggen dat ieder object verantwoordelijk is voor één kerntaak en *loose coupling* wil zeggen dat de objecten onderling losgekoppeld zijn en o.a. makkelijk uitwisselbaar zijn.

De opdrachtgever levert aan de poetsdienst dus de werkmiddelen. Dit kan op twee manieren gebeuren: ofwel d.m.v. een argument van de constructor ofwel door middel van een *setter*. Dit doorgeven van afhankelijke objecten noemt met *dependency injection*. M.a.w. *inversion of control* wordt gerealiseerd door middel van *dependency injection*.

We nemen even de code erbij:

```
public interface CleaningTool {
   public void doCleanJob();
}
```

```
public class Broom implements CleaningTool {
   public void doCleanJob() {
      System.out.println("Scrub scrub");
   }
}
```

```
public class VacuumCleaner implements CleaningTool {
   public void doCleanJob() {
      System.out.println("Zuuuuuuuuuuu");
   }
}
```

```
public interface CleaningService {
   public void clean();

}
```

```
public class CleaningServiceImpl implements CleaningService {
   private CleaningTool tool;

   public void setCleaningTool(CleaningTool tool) {
      this.tool = tool;
   }

   public void clean() {
      System.out.println("Cleaning the house");
      tool.doCleanJob();
   }
}
```

Het hoofdprogramma:

```
public class HouseApp {
   public static void main(String[] args) {
      CleaningTool broom = new Broom();
      CleaningTool vacuum = new VacuumCleaner();

      CleaningServiceImpl jill = new CleaningServiceImpl();
      jill.setCleaningTool(broom);
      CleaningServiceImpl jane = new CleaningServiceImpl();
      jane.setCleaningTool(vacuum);

      jill.clean();
      jane.clean();
   }
}
```

We hebben meteen een team ingehuurd waarbij de ene met de borstel en de andere met de stofzuiger aan de slag kan.

Opdracht 1: *Inversion Of Control*

In deze opdracht gaan we bovenstaand voorbeeld uitproberen.

- Maak een nieuw project in je IDE, bij voorkeur een *Maven*-project. Voorlopig hoeft *Spring* nog niet toegevoegd te worden aan het *classpath*, of als *dependency*.
- Maak de interfaces `CleaningTool` en `CleaningService`.
- Maak de klassen `Broom` en `VacuumCleaner`.
- Optioneel: maak een klasse `Sponge` die de interface `CleaningTool` implementeert.
- Maak de klasse `CleaningServiceImpl`.

- Maak een hoofdprogramma `HouseApp` waarin een aantal mensen aan het werk gezet worden om het huis schoon te maken, elk met hun eigen poetsgerei.

1.3.2 BeanFactory en ApplicationContext

Door de *Inversion of Control* hebben we herbruikbare en herconfigureerbare componenten gemaakt. Bovendien hebben we gebruik gemaakt van de *JavaBean*-specificatie met zijn *getters* en *setters* om communicatie met de objecten op een gestandaardiseerde wijze te laten verlopen. Kort samengevat komt de *JavaBean*-specificatie hier op neer:

1. Objecten hebben minstens een standaard constructor zonder argumenten zodat ze op makkelijke manier geïnstantieerd kunnen worden.
2. Properties (gegevens of afhankelijkheden) kunnen ingesteld en opgevraagd worden met *setters* en *getters*, ook wel *accessor methods* genoemd. Deze zien er als volgt uit:

```
public datatype getMyProperty()
public void setMyProperty(datatype prop)
```

De namen van de *properties* volgen de *camelcase*-notatie: d.w.z. dat ieder nieuw woord begint met een hoofdletter.

Objecten die aan deze voorwaarden voldoen, noemen we *JavaBeans* of ook gewoon *beans*.

De afhankelijkheden worden niet langer door de *bean* zelf gemaakt, maar worden van buiten aangereikt. We hebben daardoor enerzijds meer flexibiliteit bij het inzetten van de componenten maar anderzijds dienen deze afhankelijkheden wel eerst op de juiste wijze geïnjecteerd te worden alvorens we de *beans* kunnen gebruiken.

We zien dit in onze hoofdapplicatie:

```
CleaningTool broom = new Broom();
CleaningTool vacuum = new VacuumCleaner();
CleaningTool sponge = new Sponge();

CleaningServiceImpl jill = new CleaningServiceImpl();
jill.setCleaningTool(broom);
CleaningServiceImpl jane = new CleaningServiceImpl();
jane.setCleaningTool(vacuum);
CleaningServiceImpl richard = new CleaningServiceImpl();
richard.setCleaningTool(sponge);

jill.clean();
jane.clean();
richard.clean();
```

We hebben met dit concept dus meer werk met het instantiëren en aaneenrijgen (*wiring*) van de *beans* alvorens we ze kunnen gebruiken. Deze acties moeten dus ook weer ergens in de software gebeuren. In ons voorbeeld heeft de opdrachtgever eerst heel wat werk voor hij zijn poetsdiensten aan het werk kan zetten. Dit kan uiteraard niet de bedoeling zijn.

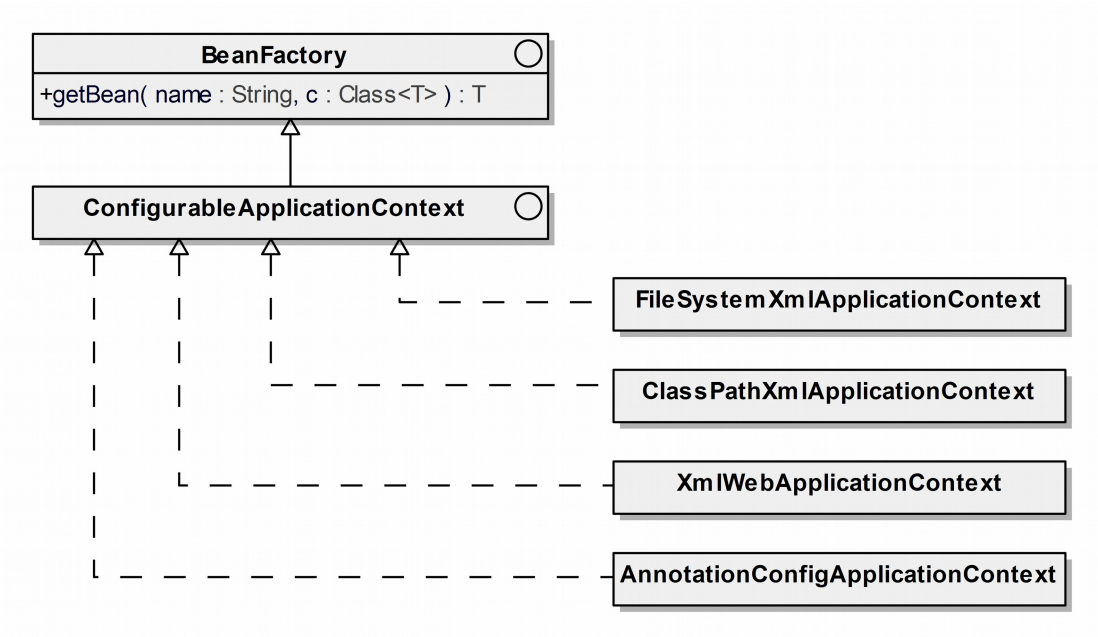
En hier komt het *Spring framework* ons ter hulp. *Spring* neemt het instantiëren en aaneenrijgen van de *beans* voor zijn rekening en levert ons kant en klare *beans* af die onmiddellijk gebruikt kunnen worden. Dit alles wordt afgehandeld door de `BeanFactory`. Deze leest de nodige configuratiegegevens uit een externe bron, instantieert de nodige *beans* en rijgt ze vervolgens op de juiste wijze aaneen.
Deze `BeanFactory` is een implementatie van het *design pattern* ***Factory.***

Een speciale afgeleide interface van de `BeanFactory` is de `ConfigurableApplicationContext`. Deze biedt nog extra mogelijkheden waar we in een applicatie handig gebruik van kunnen maken:

- Het gebruik van internationalisatie (I18N).
- Het laden van allerlei *resources*.
- Het afvuren van *events* naar *beans*.

Doorgaans wordt steeds de `ConfigurableApplicationContext` in een applicatie gebruikt i.p.v. de `BeanFactory`. We zullen later in de cursus de extra mogelijkheden hiervan onder de loep nemen.

De interface `ConfigurableApplicationContext` heeft veel implementaties. In onderstaand schema geven we er enkele:



Afbeelding 9: BeanFactory en implementaties

De configuratiegegevens die de `ConfigurableApplicationContext` nodig heeft voor het instantiëren en aaneenrijgen van de *beans* kunnen afkomstig zijn van twee bronnen:

1. **XML**: De configuratiegegevens zijn beschreven in een XML-document. Dit kan eventueel uitgebreid worden met annotaties in de *bean*-klasse.
2. **Java-configuratie**: Sinds *Spring 3.0* kan de configuratie ook volledig gedaan worden in configuratie-objecten die voorzien zijn van annotaties.

In bovenstaande afbeelding zien we vier implementaties[1] van de interface `ConfigurableApplicationContext`:

1 Voor een volledig overzicht verwijzen we naar de *Spring* API-documentatie.

Implementatie	*Omschrijving*
`FileSystemXml ApplicationContext`	De configuratie komt uit een **XML**-bestand dat zich op het bestandssysteem bevindt. We moeten hierbij het absolute of relatieve pad van het XML-bestand opgeven.
`ClassPathXml ApplicationContext`	De configuratie komt uit een **XML**-bestand dat zich in het *classpath* van de applicatie bevindt. We moeten hier het relatieve pad binnen het *classpath* opgeven. We kunnen hierdoor het XML-bestand in een JAR-bestand plaatsen.
`XmlWeb ApplicationContext`	De configuratie komt uit een **XML**-bestand dat zich in het WAR-bestand van een webapplicatie bevindt.
`AnnotationConfig ApplicationContext`	De configuratie komt van **Java-configuratie-objecten** die voorzien zijn van annotaties.

Tabel 1: Implementaties van ApplicationContext

Aanvankelijk werd binnen *Spring* vooral gebruik gemaakt van XML. Het gebruik van XML heeft echter een aantal nadelen: fouten in de configuratie komen vaak pas bij de uitvoering (*at runtime*) aan het licht. Dat komt o.a. omdat klassenamen en namen van *properties* in de tekst van het XML-bestand gedefinieerd zijn. Indien daar bijvoorbeeld een syntaxfout in zit, zal die niet door de compiler gedetecteerd worden maar zal die pas bij het interpreteren van het XML-bestand ontdekt worden. Een tweede nadeel is dat we bij de XML-configuratie geen gebruik kunnen maken van bijvoorbeeld *code completion* om een *bean* te configureren. We moeten hier in de documentatie opzoeken wat de volledige naam van een klasse is en welke eigenschappen voorhanden zijn voor de configuratie.

Kortom, alle voordelen die we hebben met het louter werken binnen Java-code vallen weg: controle van de syntax door de compiler, *code completion*, koppeling met de JavaDoc enz...

Om aan dit euvel tegemoet te komen is een alternatieve configuratie ontwikkeld die geen gebruik maakt van XML maar waarbij de configuratie volledig met Java-code en annotaties gebeurt. Dit is de `AnnotationConfigApplicationContext`. Doorgaans spreekt men gewoon van *JavaConfig*.

Vermits de laatste jaren *JavaConfig* meer gebruikt wordt dan XML, zullen we in deze cursus in eerste instantie gebruik maken van deze op Java gebaseerde configuratie.

De instructies voor het aaneenrijgen van de *beans* worden beschreven in een afzonderlijke Java-klasse die voorzien is van een aantal annotaties.

We zullen dit alles illustreren aan de hand van ons huishoudvoorbeeld.

We maken een nieuwe klasse waarin we de instantiatie en configuratie van de verschillende *beans* onderbrengen:

```
@Configuration
public class AppConfig {

   @Bean
   public CleaningTool broom() {
      return new Broom();
   }

   @Bean
```