Methodology

Domain Driven Design beschrijft hoe complexe softwaresystemen ontworpen kunnen worden op basis van een businessdomein. Het helpt met het vastleggen van een complex businessdomein in een software-implementatie.. Een direct verband tussen het businessdomein en de softwarekenmerken resulteert in simpel te begrijpen en goed uit te breiden systemen.

De voordelen van Domain Driven Design

Tijd om ideeën te adopteren en te profiteren

Ondanks dat de ideeën achter Domain Driven Design niet nieuw zijn en de voordelen voor de hand liggen, wordt er bij veel software-implementaties voor een klassiek design gekozen. De voornaamste reden hiervoor schuilt in de technische beperkingen die jarenlang door frameworks zijn opgelegd.

Het is een groot probleem voor projecten wanneer er binnen een team verschillende termen worden gebruikt voor belangrijke businessconcepten. Wanneer technische leden van het team een vertaalslag moeten maken van het jargon van een domeinexpert naar de termen die in de software gebruikt worden, gaat er veel belangrijke informatie verloren.

Een oplossing voor dit probleem is het ontwikkelen van een gemeenschappelijke taal rondom het domeinmodel. Domeinexperts moeten bezwaar maken wanneer delen van deze taal niet logisch zijn, omdat dit betekent dat de taal het domein niet goed beschrijft.

De gemeenschappelijke taal moet worden gebruikt in de code en voor elke vorm van communicatie binnen het team, wat zorgt voor een goede connectie tussen de software en het businessdomein. Het ontwikkelen van een gemeenschappelijke taal is een zeer belangrijk deel van Domain Driven Design. In de rest van dit artikel ligt de focus echter op de implementatie van het domein en zal de gemeenschappelijke taal niet terugkeren.

Applicatielagen

Domain Driven Design beschrijft de volgende vier applicatielagen.

User Interface Layer

Zoals de naam al aangeeft is de User Interface-laag verantwoordelijk voor het tonen van informatie aan een gebruiker en het interpreteren van gebruikerscommando's.

Application Layer

In de klassieke designs bevat deze laag het merendeel van alle businesslogica in de vorm van services. Bij Domain Driven Design is dit niet het geval. Deze laag bevat geen businessregels of -logica en is verantwoordelijk voor het coördineren van taken en delegeren van werk aan de domein laag.

Domain Layer

Deze laag is het hart van de businesssoftware en is verantwoordelijk voor de representatie van business concepten en regels. Dit is de belangrijkste van de vier lagen en daarom ook het onderwerp van de rest van dit artikel.

Infrastructure Layer

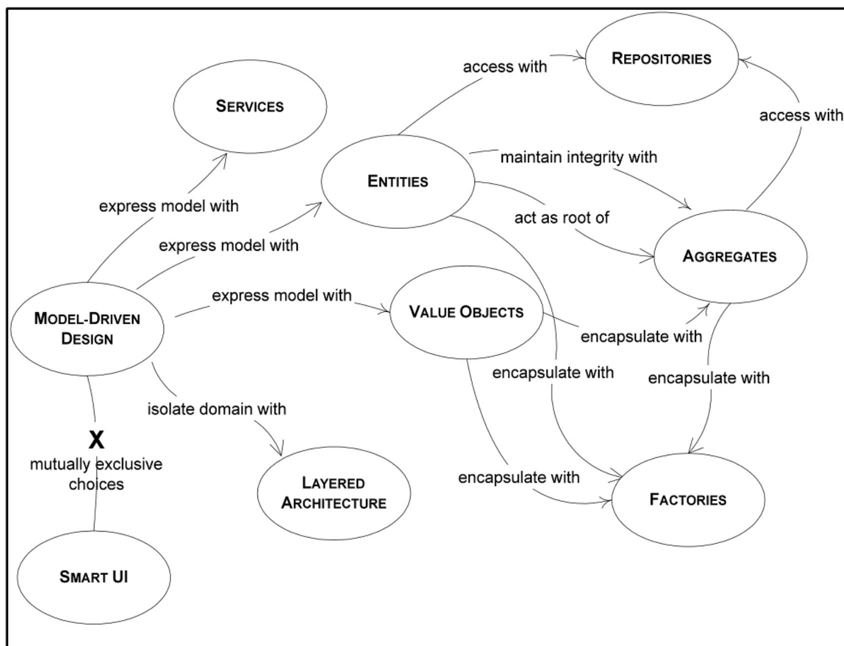
De onderste laag van de vier levert technische services aan de bovenliggende lagen zoals het versturen van berichten en persisteren van het domein.

Een voorbeeld

Om te illustreren hoe de domeinlaag van een applicatie is opgebouwd, gebruiken we het volgende versimpelde businessdomein uit de bancaire wereld.

Een klant (Customer) heeft één of meerdere rekeningen (Accounts) en kan geld overmaken van één van die rekeningen naar een andere rekening binnen het systeem. Wanneer er geld wordt overgemaakt, dan moet deze transactie (Transaction) opgeslagen worden in het systeem.

Ronald van Rijn is werkzaam als software architect bij IPROFS.



Figuur 1: het businessdomein.

Bouwstenen van het domein

In de klassieke designs is de domeinlaag weinig meer dan een verzameling dataobjecten. Deze objecten bevatten enkel de domeininformatie en het domeingedrag wordt door andere lagen toegevoegd.

Bij Domain Driven Design wordt de domeinlaag opgebouwd uit een aantal bouwstenen die tezamen zowel de informatie als het gedrag van het business-domein omvatten.

Entities

Entities worden onderscheiden op basis van identiteit. Deze identiteit verandert niet tijdens de levensduur van een applicatie en zorgt voor continuïteit.

In ons voorbeelddomein kunnen we twee entities onderscheiden, een Customer en een Account. Customers worden onderscheiden op basis van een identiteit, bijvoorbeeld customer id. Elk attribuut van een Customer kan veranderen, maar zolang het customer id hetzelfde blijft, weten we welke Customer het is. Het is daarom belangrijk bij het implementeren van entities dat deze identiteit niet kan wijzigen.

Iedereen heeft wel een keer de volgende implementatie gezien van een domein object. Het object wordt volgens de JavaBeans specificatie geïmplementeerd met een private property en een public getter en setter voor alle velden, zodat het daarna gebruikt kan worden door een ORM framework.

```
public class Customer {
   ...
   private Long id;

   public Long getId() {
       return id;
   }
```

```
   public void setId(Long id){
       this.id = id;
   }
   ...
}
```

Deze implementatie geeft gebruikers van het Customer object de mogelijkheid om het id te wijzigen, waardoor de identiteit van dit object verloren zou gaan.

Voor de huidige ORM frameworks is het al lang geen noodzaak meer om voor alle velden een getter en een setter te specificeren. Onderstaande implementatie voorkomt dat de identiteit verloren kan gaan en is dus een betere implementatie van een entity.

```
public class Customer {
   ...
   private Long id;

   public Long getId() {
       return id;
   }
   ...
}
```

Informatie en gedrag

Entities bevatten niet enkel de informatie van een domeinconcept, maar ook het gedrag. In huidige software-implementaties worden deze vaak gescheiden in domein objecten en services.

De volgende implementatie is niet ongewoon. We hebben een domeinobject voor Customer en voor het gedrag bij dit object hebben we een bijbehorende service. Om het wachtwoord van Customer te veranderen, moet de changePassword method op de service worden aangeroepen. Deze methode gaat vervolgens controleren of het nieuwe wachtwoord nog niet eerder is gebruikt en of deze aan alle regels voldoet en zal het nieuwe wachtwoord daarna aan het Customer object toevoegen.

```
public class CustomerService {
   ...
   public void changePassword(
       String password){
      //change password logic
      //here
   }
   ...
}
```

Het veranderen van het wachtwoord is de natuurlijke verantwoordelijkheid van het Customer object. Een implementatie waarbij de methode voor het wijzigen van het wachtwoord op het Customer object zit, ligt daarom veel dichter bij het domein dat we proberen te implementeren.

```
public class Customer {
   ...
   private Long id;

   public Long getId() {
       return id;
   }
```

```
    public void
        changePassword(
        String password) {
       //change password logic
       //here
    }
    ...
}
```

Het is goed mogelijk dat het Customer object andere classes nodig heeft om de logica voor het wijzigen van een wachtwoord te implementeren. Er zou bijvoorbeeld een wachtwoord validatieclass in ons systeem kunnen zijn dat hiervoor nodig is. Later in het artikel wordt gedemonstreerd hoe het Customer object toegang kan krijgen tot andere classes.

Value Objects

Value Objects worden onderscheiden op basis van hun attributen. Ze vertegenwoordigen een waarde en worden beschouwd als onveranderlijk.
Transaction is een voorbeeld van een value object binnen ons domein. Wanneer er een bedrag overgeboekt wordt naar een Account wordt er een Transaction voor deze overboeking aangemaakt. Wanneer het geld is overgeboekt mag de Transaction niet meer gewijzigd worden en deze wordt dus als onveranderlijk geïmplementeerd.

```
public class Transaction {
   private final BigDecimal
      amount;
   private final Date date;

   public Transaction(
        BigDecimal amount,
        Date date) {
      this.amount = amount;
      this.date = date == null
        ? null : date.clone();
   }

   public BigDecimal
        getAmount() {
      return amount;
   }

   public Date getDate() {
      return date == null ?
         null : date.clone();
   }
}
```

Het is ook hier weer belangrijk dat het Transaction object zijn interne properties goed encapsuleert, zodat deze niet gewijzigd kunnen worden door andere classes.

Aggregates

Een aggregate is een cluster van gerelateerde objecten die we als één geheel beschouwen voor het wijzigen van data. Elke aggregate heeft een root entity en creëert een duidelijke grens rondom de gerelateerde objecten. Een aggregate controleert alle toegang op de objecten binnen die grens.

Account is binnen ons domein de root van een aggregate. Voor elke overboeking van geld wordt een Transaction in Account bijgehouden. Dit zou op de volgende manier geïmplementeerd kunnen worden.

```
public class Account {
   ...
   private List<Transaction>
        transactions;

   public List<Transaction>
        getTransactions() {
      return transactions;
   }

   public void setTransactions
      (List<Transaction>
        transactions) {
      this.transactions =
         transactions;
   }
   ...
}
```

Echter, wanneer we de aggregate op deze manier zouden implementeren, verliest deze alle controle over de gerelateerde objecten binnen zijn grenzen. De lijst met Transactions kan door een andere class opgevraagd worden en deze kan er vervolgens van alles mee doen.
De volgende implementatie behoudt alle controle over de lijst met Transactions door deze goed te encapsuleren. Andere classes, die een lijst van alle Transactions van Account willen hebben, krijgen een lijst terug die niet gewijzigd kan worden en wanneer er een Transaction toegevoegd moet worden dan heeft de aggregate hierover alle controle.

```
public class Account {
   ...
   private List<Transaction>
        transactions;

   public List<Transaction>
        getTransactions() {
      return Collections
         .unmodifiableList(
             transactions);
   }

   public void addTransaction(
        Transaction
           transaction) {
      transactions
         .add(transaction);
   }
   ...
}
```

Services

Services bevinden zich in meerdere lagen van een applicatie. Veel services zijn technisch van aard en bevinden zich in de infrastructuur laag, maar er zijn ook services die deel uitmaken van het domein.

Wanneer een significant proces niet op een natuurlijke manier de verantwoordelijkheid is van een entity of value object, dan wordt deze ondergebracht in een service in de domeinlaag.
Een voorbeeld hiervan is het overmaken van geld tussen twee Accounts. Dit proces kan niet worden ondergebracht bij één Account entity, geen van beide Accounts is immers volledig verantwoordelijk voor de volledige transactie. Het overmaken van geld is echter wel een zeer belangrijk concept in het bancaire domein en zal dus als service aan het domein toegevoegd worden.

Value objects hebben een waarde en zijn niet te veranderen

Het syteem is eenvoudig te begrijpen en goed te onderhouden

```
public class
    FundsTransferService {
   ...
   public void transferFunds(
       Account from,
       Account to,
       BigDecimal amount) {
      ...
   }
   ...
}
```

Factories

Het creëren van een object kan een complexe operatie zijn, maar dergelijke operaties zijn niet de verantwoordelijkheid van het gecreëerde object.
De onderstaande Account implementatie heeft een constructor waarmee een nieuwe Account instantie gecreëerd kan worden. Deze constructor kan zeer complex zijn en zal dan logica bevatten die niet thuis hoort in dit object.

```
public class Account {
   ...
   public Account(
       Customer customer,
       Country country) {
      //complex creation
      //logic here
   }
   ...
}
```

Een factory is verantwoordelijk voor het creëren van een type object en is daarom de perfecte plaats om deze complexe creatie logica neer te leggen.

```
public class AccountFactory {
   public Account
       createAccount(
       Customer customer,
       Country country) {

      //complex creation
      //logic here

      return account;
   }
}
```

Repositories

Om gebruik te maken van een object moet je een referentie naar dat object hebben. Een repository kan gebruikt worden om een referentie te krijgen naar een persistent object.
Repositories creëren de illusie van een in-memory collectie van objecten van een bepaald type. Er worden methoden beschikbaar gesteld om objecten aan deze collectie toe te voegen of eruit te verwijderen en deze methoden verbergen de operaties op een persistente bron.

```
public class AccountRepository {
   private EntityManager
       entityManager;

   public void add(Account
       account) {
      entityManager
          .persist(account);
   }

   public Account find(
       Long id) {
      return entityManager
         .find(Account.class,
            id);
   }
}
```

Dependency injection

Omdat het domein zowel eigenschappen en het gedrag van het business domein bevat kan het zijn dat objecten in het domein logica nodig heeft die in andere classes geïmplementeerd is.
Voor entities en aggregates is dit een probleem. Deze objecten worden middels repositories uit een persistente bron gehaald en zijn dus niet tijdens het opstarten van het systeem voor handen om andere classes op te injecteren.
In het Spring Framework is hier een goede oplossing voor. De @Configurable annotatie zorgt ervoor dat Spring andere classes kan injecteren wanneer er een nieuwe instantie van de geannoteerde class gemaakt wordt.
We hebben eerder een changePassword method aan het Customer object toegevoegd. De logica voor het valideren van een nieuw wachtwoord is ondergebracht in een aparte validatie class, zodat deze kan worden hergebruikt in andere delen van het systeem.
Met de @Configurable annotatie kan op het Customer object de PasswordValidator worden geïnjecteerd, zodat deze vervolgens gebruikt kan worden om het nieuwe wachtwoord te valideren.

```
@Configurable
public class Customer {
   ...
   private Long id;

   @Autowired
   private PasswordValidator
      passwordValidator;

   public Long getId() {
       return id;
   }

   public void
       changePassword(
       String password) {
      passwordValidator.
         validate(password)

      //change password logic
      //here
   }
   ...
}
```

Conclusie

Domain Driven Design helpt met het vastleggen van een complex businessdomein in een software-implementatie. Wanneer er een direct verband bestaat tussen de software-implementatie en het businessdomein is het systeem eenvoudig te begrijpen en goed te onderhouden. De huidige frameworks zoals Spring en JPA leggen geen technische beperkingen meer op voor een Domain Driven implementatie. Het wordt tijd dat nieuwe software-implementaties deze ideeën adopteren om zo kunnen profiteren van alle voordelen van Domain Driven Design. «

Referenties
http://domaindrivendesign.org/