CHAPTER 1

Scheiding tussen GUI en Domain

FHICT

In dit hoofdstuk bespreken we een aantal principes bij de scheiding tussen GUI-klassen en domeinklassen (domain classes). Alvorens deze principes te presenteren leggen we eerst uit wat een domeinklasse is. We eindigen dit hoofdstuk met illustraties waar de beoogde scheiding goed of slecht (lees: minder goed) is uitgevoerd.

1.1 Domein

Elke applicatie (of app) is een hulpmiddel om zekere processen sneller, prettiger of anderszins beter te laten verlopen. Deze processen hebben betrekking op het creëren, inspecteren, wijzigen of verwijderen van informatie. Bijvoorbeeld het aanvragen van een nieuwe bankrekening, het opvragen van het saldo van een bankrekening, het overmaken van geld van de ene naar de andere bankrekening, of het afscheid nemen als klant van een bank. In deze gevallen gaat het steeds om informatie uit het applicatiedomein van financieel verkeer. De Informatie is gerelateerd aan objecten uit dit applicatiedomein: zoals Bankaccount, Bank, Client en Transaction. We noemen dit objecten van domeinklassen. Daar tegenover staan objecten die een middel vormen

- om domeinobjecten te kunnen visualiseren in een GUI, of
- om informatie in objecten te kunnen bewaren in een database of bestand, of
- om informatie in objecten over een netwerk te kunnen versturen of ...

Deze drie laatstgenoemde objecten behoren niet tot het domein.

We hebben voor het domein van financieel verkeer de domeinklassen Bankaccount, Bank, Client, Transaction geintroduceerd. Voor een spel met helden en monsters zouden de domeinklassen Hero, Monster, Player en Game een zelfde centrale rol kunnen vervullen. Deze klassen herbergen de belangrijke informatie om zo'n spel in een GUI zichtbaar te kunnen maken.

Samenvattend: Domeinklassen herbergen de relevantie informatie uit het beoogde applicatiedomein.

Challenge1: welke domeinklassen zou jij willen introduceren binnen de context van het spel galgje? Challenge2: welke domeinklassen zou jij willen introduceren binnen de context van het verzenden, inzien en verwijderen van e-mails?

1.2 Principes

Wij adviseren om je te houden aan onderstaande vier principes, ter wille van een goede scheiding tussen de programmacode in een GUI en de programmacode in de domeinklassen. Deze lijst met principes is niet compleet, maar wordt in de loop van je studie uitgebreid en verder genuanceerd.

1. Gebruik geen GUI-objecten in domeinklassen. Want dan kun je zo niet gemakkelijk overstappen naar een andere GUI (bijv. WindowsForm, Web GUI, smart phone, andere vormgeving van de GUI, verschillen in GUI per soort gebruiker). Neem binnen een domein-object ook geen plaatjes op (later wil jewel eens een ander plaatje gebruiken). Je software wordt daarmee flexibeler en beter onderhoudbaar.
2. Wees voorzichtig met het opnemen van zelf gedefinieerde reken- of validatiemethoden binnen een GUI-klasse (zie 3.)
3. Beperkingsregels (zoals het saldo van een bankrekening mag niet lager worden dan de kredietlimiet) en berekeningen (zoals hoeveel rente krijg je per maand bijgeschreven) moet je voor 100% borgen binnen de betreffende domeinklasse die over alle vereiste informatie beschikt.
4. Als je van domeinobjecten een tekstrepresentatie in de GUI wilt kunnen opnemen, dan is het handig om de ToString-methode van de betreffende domeinklasse te 'overriden'.

1.3 Illustraties

Deze vier principes illustreren we aan de hand van de bankrekening-casus.

ad 1. Fout: Binnen een ChangeBalance-methode van de Bankaccount-klasse een messagebox (MessageBox.Show) activeren zodra er wordt geprobeerd om teveel geld van het BankAccount-object af te schrijven.

```
public class Bankaccount {
  private decimal balance;
  private decimal threshold; // not negative
  ...
  public void ChangeBalance(decimal amount) {
    if (balance + amount < -threshold) {
      MessageBox.Show("withdrawal of " + amount + " is not allowed");
      } else {
      balance -= amount;
    }
  }
}
```

Goed: De ChangeBalance-methode van de Bankaccount-klasse retourneert bijvoorbeeld een string. De betreffende string representeert of het opnemen van het geldbedrag al dan niet is geaccepteerd. Vervolgens kan deze returnwaarde naderhand in de GUI binnen een MessageBox als een message worden getoond.

```
public class Bankaccount {
  private decimal balance;
  private decimal threshold;
  ...
  public string ChangeBalance(decimal amount) {
    if (balance + amount < -threshold) {
      return "withdrawal of " + amount + " is not allowed";
    } else {
      balance -= amount;
      return "withdrawal is succeeded";
    }
  }
}
```

Merk op, in plaats van de string als returnwaarde kan er ook met een simpele bool worden gewerkt.

ad 2. Fout: Binnen de GUI-klasse wordt een hulpmethode CheckValidTransaction gedefinieerd waarbinnen alvast wordt gecontroleerd of het overmaken van geld, van de ene bankrekening naar een andere bankrekening uitvoerbaar is.

```
// een methode binnen de GUI-klasse
bool CheckValidTransaction(Bankaccount ba, decimal amount) {
  if (ba.getBalance() + amount < -ba.getTreshold()) {
    return false;
  } else {
    return true;
  }
}
```

Goed: Stuur het verzoek tot overmaken door naar het betreffende Bank-

object van de bankrekening waar het geld van wordt afgeschreven. Later rapporteert het Bank-object of het overmaken is geslaagd, bijvoorbeeld door middel van een string (zie ook ad 1.)

```
public class Bank {
  ...
  public string Transfer(string from, string to, decimal amount) {
    // check if bankaccount number from exists
    Bankaccount baFrom = GetBankaccount(from);
    if (baFrom==null) {
      return "Bankaccount " + from + " does not exist.";
    } else {
      // check if bankaccount number to exists
      Bankaccount baTo = GetBankaccount(to);
      if (baTo==null) {
        return "Bankaccount " + to + " does not exist.";
      } else {
        string result =  baFrom.ChangeBalance(-amount);
        if (result == "withdrawal is succeeded") {
          baTo.ChangeBalance(amount)
        }
        return result;
      }
    }
  }
```

ad 3. Neem even aan dat er bij een bank geen twee klanten mogen voorkomen met dezelfde combinatie van naam/adres/geboortedatum. Fout: Binnen de Bank-klasse wordt er, zodra er een nieuwe klant wordt toegevoegd, niet gecontroleerd of er al een klant bestaat met dezelfde naam, adres en geboortdedatum.

```
public class Bank {
  private List<Client> clients;
  ...
  // andere gegevens van de klant zijn nu even genegeerd:
  public void AddClient(string name, string place, DateTime
    birthdate) {
    Client client = new Client(name, place, birthdate);
    clients.Add(client);
  }
}
```

Goed: Binnen een AddClient-methode van de Bank-klasse wordt eerst gecontroleerd of er al een klant bestaat met dezelfde naam, adres en geboortdedatum. Zo ja, dan wordt het nieuwe Client-object niet gecreëerd en geregistreerd; anders wel. De AddClient-methode heeft als returnwaarde het nieuwe Client-object. Als de returnwaarde null is, is er geen Client-object gecreëerd.

```
public class Bank {
  private List<Client> clients;
  ...
  // andere gegevens van de klant zijn nu even genegeerd:
  public Client AddClient(string name, string place, DateTime
     birthdate) {
    Client client = GetClient(name, place, birthdate);
    if (client != null) return null;
    client = new Client(name, place, birthdate);
    clients.Add(client);
    return client;
  }
}
```

ad 4. Binnen de GUI wordt een lijst met gegevens van alle bankrekeningen van een specifieke klant getoond. Dan is het handig als de ToString-methode van de Bankaccount-klasse wordt geherdefinieerd; bijvoorbeeld door te volstaan met het rekeningnummer en de naam van de eigenaar van de bankrekening:

```
public class Bankaccount {
  private string nr;
  private Client owner;
  ...
  public override string ToString() {
    return this.nr + ": " + this.owner.GetName();
  }
}
```

Deze ToString-methode kan vervolgens binnen de GUI worden aangeroepen.