UML: Class diagrams en sequence diagrams

Inleiding

Bij het analyseren en ontwerpen van een objectgeoriënteerde applicatie wordt in de praktijk veel gebruik gemaakt van Unified Modelling Language (UML). UML is een verzameling (diagram)technieken waarmee facetten van de analyse en het ontwerp kunnen worden beschreven. Zo gebruik je

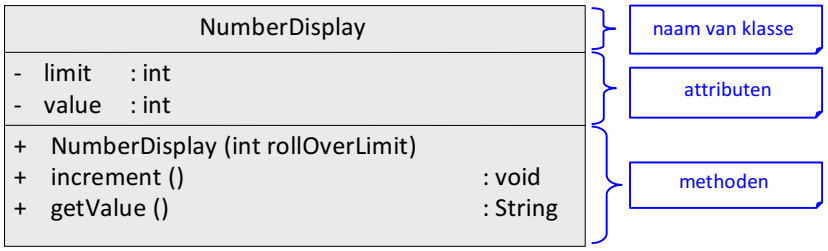
- *use cases* voor het in kaart brengen van de interactie tussen gebruikers en het systeem;
- *class diagrams* om weer te geven welke klassen een rol spelen in een OO-applicatie, welke attributen en methoden de klassen bezitten en hoe de verschillende klassen met elkaar samenhangen;
- *sequence diagrams* om te beschrijven hoe objecten met elkaar samenwerken om een taak te volbrengen.

Voorbeeld 1: Clock display

Dit voorbeeld wordt besproken in hoofdstuk 3 (§3.1 t/m §3.11) van het boek “Objects First with Java”. De broncode van de klassen NumberDisplay en ClockDisplay vind je in respectievelijk §3.6 en §3.8.

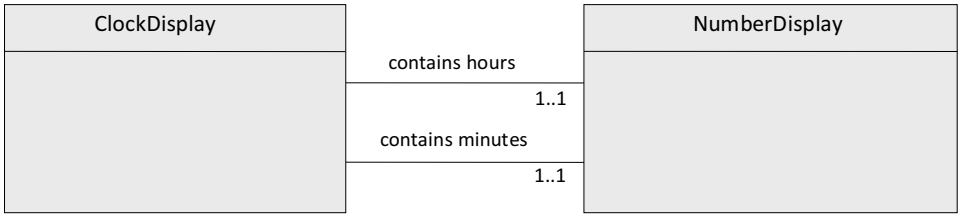
Class diagrams

In het ClockDisplay-voorbeeld spelen twee klassen een rol. Dat zijn de klassen ClockDisplay en NumberDisplay. Hieronder zie je de klasse NumberDisplay met bijbehorende attributen en methoden.



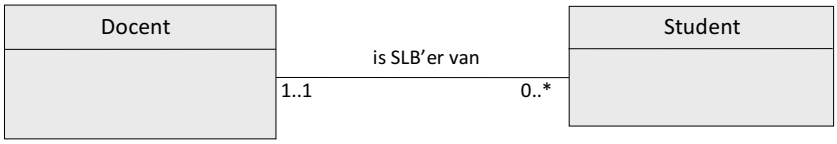
De klasse NumberDisplay heeft twee attributen van het type int, dat zijn limit en value.
NumberDisplay heeft een constructor en methoden *increment* en *getValue*. De constructor heeft een parameter *rollOverLimit* van het type int. De methode *increment* heeft geen parameter en ook geen returntype (void). De methode *getValue* heeft geen parameter en heeft String als returntype.
Voor elke attribuutnaam of methodenaam staat een + of een -; + betekent public en - private.

Elk ClockDisplay-object maakt gebruik van twee NumberDisplay-objecten: de ene voor het weergeven van de uren en de andere voor de minuten. Elk ClockDisplay-object is dus gerelateerd met twee NumberDisplay-objecten. Deze relaties, in UML spreekt men van *associaties*, geven we in het class diagram weer door een lijn te tekenen van ClockDisplay naar NumberDisplay. De ene associatie noemen we *contains hours*, de andere *contains minutes*. In het volgende class diagram zijn deze associaties weergegeven. De attributen van de klassen zijn achterwege gelaten.
In het bovenstaande diagram zie je aan de rechterkant bij de associatie *contains hours* 1..1 staan. Dat heet de *multipliciteit* en betekent: elk clockdisplay bevat precies één urendisplay. Net zo bevat elk clockdisplay precies één minutendisplay.

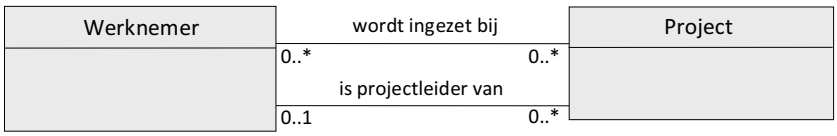


Op de volgende pagina zie je meer voorbeelden van multipliciteiten bij associaties.

Voorbeeld SLB
In het onderstaande class diagram zie een associatie *is SLB'er van* tussen de klassen Docent en Student. De multipliciteiten bij deze associatie geven aan dat een docent SLB'er is van 0 of meer studenten (0..*) en dat een student precies één docent als SLB'er heeft (1..1).

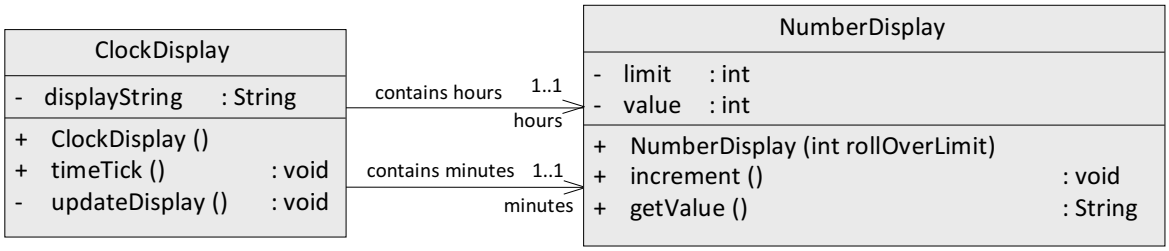


Voorbeeld Projectinzet
Een bedrijf heeft werknemers die worden ingezet bij projecten die worden uitgevoerd. Een van de werknemers speelt de rol van projectleider. Dit is in het onderstaande class diagram weergegeven.



Je ziet de associaties *wordt ingezet bij* en *is projectleider van* tussen de klassen Werknemer en Project. De multipliciteiten bij de associatie *wordt ingezet bij* geven aan dat een werknemer bij 0 of meer (0..*) projecten wordt ingezet en dat er 0 of meer werknemers meewerken aan een project (0..*). De multipliciteiten bij de associatie *is projectleider van* geven aan dat een werknemer 0 of meer (0..*) projecten kan leiden en dat een project een of (nog) geen (0..1) projectleider heeft.

Hier volgt een meer uitgebreide versie van het class diagram bij het ClockDisplay-programma:



We bekijken twee facetten van dit *class diagram*:

- *rollen en rolnamen*
  De klassen ClockDisplay en NumberDisplay spelen allebei een rol in de associatie *contains hours*.
  Het uiteinde van een associatie wordt dan ook wel een *rol* genoemd.
  De rol die NumberDisplay in de associatie *contains hours* speelt wordt *hours* genoemd.
  De rol die NumberDisplay in de associatie *contains minutes* speelt noemen we *minutes*.
- *navigatie*
  Bij beide associaties wijst een pijl richting de klasse NumberDisplay. De associatie *contains hours* loopt van ClockDisplay naar NumberDisplay. Dit heet *navigatie* en betekent voor deze situatie dat een instantie van ClockDisplay de bijbehorende instantie van NumberDisplay (*hours*) kent en 'm een opdracht kan geven door een methode aan te roepen. Omgekeerd kent een instantie van NumberDisplay niet de ClockDisplay-instantie waar hij bij hoort.

De java-code van de klasse ClockDisplay, die correspondeert met het bovenstaande class diagram, ziet er als volgt uit:

```
public class ClockDisplay
{
    private NumberDisplay hours;
    private NumberDisplay minutes;
    private String displayString;

    public ClockDisplay()
    {
        ……
    }

    public void timeTick()
    {
```

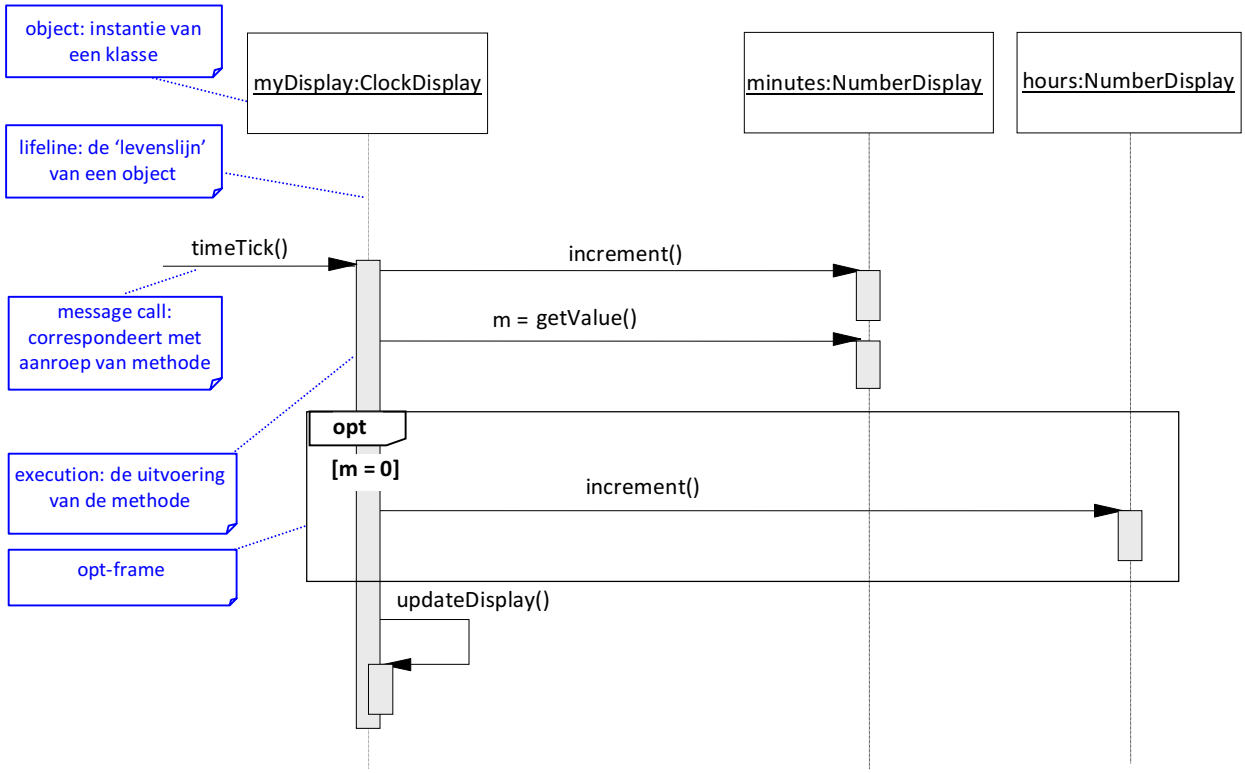
```
        ……
    }

    private void updateDisplay()
    {
        ……
    }
}
```

Je ziet in de Java-code dat de klasse ClockDisplay twee attributen heeft van het type NumberDisplay: *hours* en *minutes*. Het attribuut *hours* correspondeert met de gelijknamige navigeerbare rol (d.w.z. de rol aan de pijlkant) van de associatie *contains hours*. Net zo correspondeert het attribuut minutes met de rol *minutes* van de associatie contains *minutes*.

Tot slot van dit voorbeeld nog de volgende opmerking: In een class diagram is het type van een attribuut nooit gelijk aan een klasse in hetzelfde diagram. Een relatie tussen twee klassen beschrijf je altijd door gebruik te maken van een associatie.

Sequence diagram

In een sequence diagram wordt weergegeven hoe objecten elkaar boodschappen (*messages*) sturen en op basis daarvan opdrachten uitvoeren om zo samen een taak te volbrengen.
In het onderstaande sequence diagram zie je het object *myDisplay* van het type ClockDisplay en de objecten *minutes* en *hours* van het type NumberDisplay. Het object *myDisplay* maakt gebruikt van *minutes* en *hours*. Onder elk van deze objecten zie je een verticale stippellijn, dat is de *life line* van het betreffende object.
In het sequence diagram wordt getoond wat er gebeurt wanneer bij het object *myDisplay* de methode *timeTick* wordt aangeroepen.



Het proces start wanneer de methode *timeTick* wordt aangeroepen bij het object *myDisplay*, zie de pijl aan de linkerkant van het diagram met de naam van de betreffende methode erbij. Dit wordt een *message call* genoemd. Aan het uiteinde van de pijl begint een dunne verticale balk; deze balk geeft de uitvoering van de betreffende methode aan en wordt ook wel *execution bar* genoemd.

Bij het uitvoeren van de methode *timeTick* gebeurt het volgende:

- *myDisplay* roept om te beginnen de methode *increment* aan bij *minutes*.
- Vervolgens roept *myDisplay* de methode *getValue* aan bij het object *minutes*. Dit is een message call met een returnwaarde, die wordt opgeslagen in de variabele *m*.
- Daarna wordt gecontroleerd om m gelijk is aan 0. Zo ja, dan wordt de methode increment aangeroepen bij het object *hours*. Er wordt gebruik gemaakt van een *opt-frame*, daarmee wordt een keuzeopdracht weergegeven.

- Tenslotte roept *myDisplay* bij zichzelf de methode *updateDisplay* aan. Dit heet een *self message call*.

Ga zelf na dat de in het sequence diagram beschreven acties corresponderen met de implementatie van de methode *timeTick* in Java.

```
public void timeTick() {
   minutes.increment();
   if (minutes.getValue() == 0) {
      hours.increment();
   }
   updateDisplay();
}
```