Associaties

Hoofdstuk 9

inleiding

- ❏ Wat hebben we reeds geleerd?
  - ❏ Objecten brengen gegevens en gedragingen samen
- ❏ Variabelen en codeblokken
- ❏ OOP is zenden van boodschappen naar objecten
- ➔ Wat is het nut daar nu van?
  - ◆ Code reuse

Als we een **klasse goed gedefinieerd** hebben, kunnen we deze **hergebruiken** in een andere context

Addendum aan de inleiding

- Code reuse:
  - ➔ Relaties leggen tussen klassen.
  - ➔ Dus relaties leggen tussen objecten

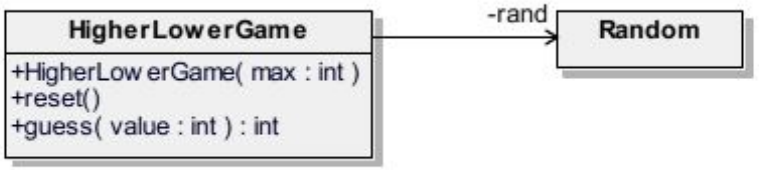
In een **relatie** hergebruiken we code die reeds in een andere klasse geschreven is.

We gaan verder in op de **verschillende vormen** van relaties.

Associaties

- Een **relatie tussen klassen** of **objecten** wordt ook wel een **associatie** genoemd.
- Het ene object maakt gebruik van de mogelijkheden van een ander object.

Hoger – Lager



Afbeelding 85: UML-diagram van een associatie

- De navigeerbaarheid is hier **unidirectioneel.**
  - ➔ word aangegeven door de pijl in UML diagram.
- We hebben de klasse HigherLowerGame en de bestaande klasse Random.
- De eerste gebruikt de functionaliteit van de tweede. In de eerste klasse is er daarom een referentie naar een object van de tweede.

ctrl+c => ctrl+v

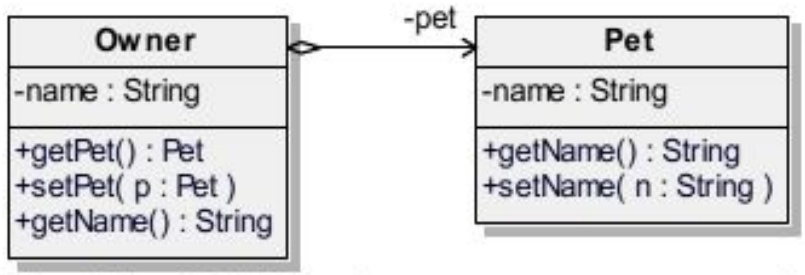
```
package examples.associations;
import java.util.*;

public class HigherLowerGame {
   private int value;
   private int max;
   private Random rand;

   public HigherLowerGame(int max) {
      this.max = max;
      rand = new Random();
      reset();
   }

   public void reset() {
      value = rand.nextInt(max + 1);
   }

   public int guess(int guessValue) {
      if (guessValue < value) {
         return -1;
      } else if (guessValue > value) {
         return 1;
      } else {
         return 0;
      }
   }
}
```

Aggregaties

In de vorige oefening maakte de klasse HigherLowerGame gebruik van de klasse Random.

- Soms is de relatie nauwer
  - ➔ Soms is een object de eigenaar van een ander object
    - Patiënt en zijn medische fiche
    - Eigenaar en zijn huisdier
- In bovenstaande gevallen spreken we van een aggregatie.

Aggregatie vb



Afbeelding 86: UML-diagram van een aggregatie

- In het UML diagram wordt een aggregatie aangeduid met een **open ruit**.
- Bij een aggregatie zijn beide objecten **niet onlosmakelijk** aan elkaar **gekoppeld**.
- Zowel de eigenaar als het huisdier **kunnen los van elkaar bestaan**.

But how does the code look?

Pet.java

```
package examples.associations;

public class Pet {
   private String name ="";

   public Pet() {
   }

   public Pet(String name) {
      this.name = name;
   }

   public String getName() {
      return name;
   }

   public void setName(String name) {
      this.name = name;
   }

}
```

And what about the owner?

Owner.java

```
package examples.associations;

public class Owner {
   private String name;
   private Pet pet;

   public Owner(String name) {
      this.name = name;
   }

   public String getName() {
      return name;
   }

   public Pet getPet() {
      return pet;
   }

   public void setPet(Pet pet) {
      this.pet = pet;
   }
}
```

Composities

Soms zijn twee objecten **onlosmakelijk** met elkaar **verbonden**.

➔ Het ene kan niet bestaan zonder het andere.

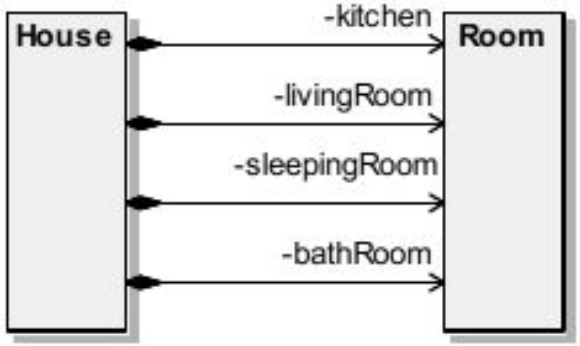
➔ Als het ene object ophoudt te bestaan is ook het andere gedoemd op te houden te bestaan.

Als voorbeeld nemen we een **huis** en zijn **kamers**.

➔ Als het huis wordt afgebroken zal ook de kamer ophouden te bestaan.

➔ In een UML diagram wordt een compositie aangeduid met een **volle ruit**.

UML-diagram



Afbeelding 87: UML-diagram van een compositie

Converted to code.

House.java

```
package examples.associations;

public class House {
   private Room kitchen = new Room("kitchen");
   private Room bathRoom = new Room("bathroom");
   private Room livingRoom = new Room("livingroom");
   private Room sleepingRoom = new Room("sleepinroom");

   public Room getKitchen() {
      return kitchen;
   }

   public Room getBathroom() {
      return bathRoom;
   }

   public Room getBathRoom() {
      return bathRoom;
   }


   public Room getLivingRoom() {
      return livingRoom;
   }


   public Room getSleepingRoom() {
      return sleepingRoom;
   }
```

High Cohesion

- We weten al dat bij het ontwikkelen van klassen we ervoor moeten zorgen dat ze zo ontworpen zijn dat ze makkelijk opnieuw kunnen worden gebruikt.
  - ➔ Hiervoor dienen we ons te houden aan het concept van de high cohesion of hoge cohesie.
- Dat komt erop neer dat een klasse gericht moet zijn op één kerntaak.