CHAPTER 1

Hoe programmeer je met Objecten?

FHICT

1.1 Hero tegen Monsters

Hoe groter software wordt, hoe tijdrovender het testen en onderhouden. Daarom wordt in de softwarewereld gezocht naar manieren om programma's onderhoudbaar te maken. Een van de meer succesvolle manieren is het werken met *objecten*.

Een team dat een Computer Game maakt over een *hero* (held) die tegen *monsters* vecht zal liefst op één plek willen programmeren wat de eigenschappen

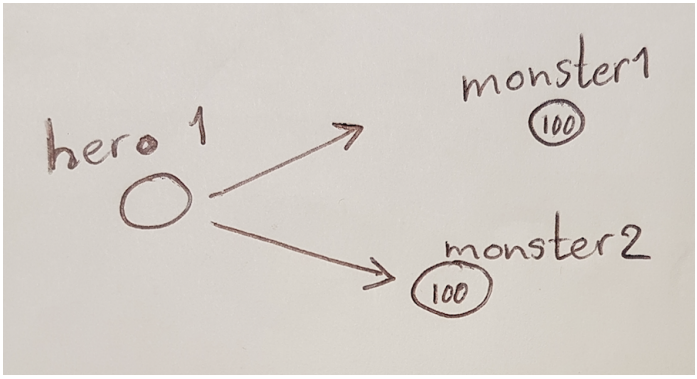

Figure 1.1 MonsterGame

van de hero zijn en ook wat een monster is en wat je aan het monster kunt vragen.

Als er 2 monsters zijn worden er 2 *objecten* gemaakt, als er 8 monsters zijn worden er 8 *objecten* gemaakt. Elk van deze objecten representeert 1 monster. De eigenschappen en het gedrag van een monster wordt geprogrammeerd in een stuk van het programma dat we een `Class` (klasse) noemen en dat (in dit voorbeeld) de naam **Monster** krijgt.

Voorbeeld in C# (hoe je dit in Visual Studio kunt doen komt een stukje verder):

```
class Monster {
      ...
}
```

waarbij op de plaats van de puntjes de code voor deze `class` komt.

Voorbeeld in Java:

```
class Monster {
      ...
}
```

Voorbeeld in Swift:

```
class Monster {
      ...
}
```

Inderdaad, deze voorbeelden zijn hetzelfde. Je zult merken dat er echt wel verschillen zijn hoe je in de ene of de andere taal een `class` noteert, maar in welke taal je ook zit: *welke* `classes` je aanmaakt blijft hetzelfde!

> **Note** In veel programmeertalen is afgesproken dat de naam van een `class` met een hoofdletter begint.

Software Engineers bedenken in het begin van een project welke `objecten` er nodig zijn en daaruit volgt welke `classes` er geprogrammeerd gaan worden. Dit kun je grotendeels bedenken zonder te weten in welke taal de software gebouwd gaat worden! Je kunt dat enigszins vergelijken met het bouwen van een huis: Waar de muren, ramen en deuren komen (de structuur) kun je tot zekere hoogte bedenken en tekenen zonder te weten of het huis met bakstenen, van beton of van hout gebouwd gaat worden!

Een `object` kan bepaalde eigenschappen hebben. Zo zal elk Monster in eerste instantie helemaal gezond zijn. Als de *hero* hem aanvalt zal het *monster* moe worden of gewond raken en uiteindelijk wellicht bezwijken.

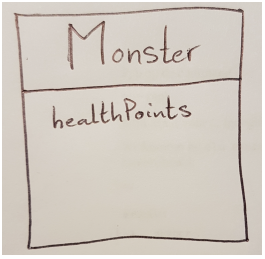

Figure 1.2 Class Monster

Hoe maak ik een class aan in Visual Studio?

Klik met rechtermuisknop op het project en kies `Add Item`, kies daarna een `class`. Onderin het scherm kun je de gewenste `class name` aangeven (`file name` is `class name` met `extensie` *.cs*) en dan op de *OK*-knop.

1.2 Gezondheid

In dit spel kunnen we dat realiseren door het monster een getal *hitPoints* te geven: Voor een pas *aangemaakt* monster staat dit op 100, bij verwondingen wordt dit gehele getal steeds kleiner, bij 0 valt het monster verslagen neer. Een waarde als *hitPoints* die elk `object` van een bepaald `type` met zich meedraagt noemen we een `Field` We maken hiertoe in Monster een `veld` *levenspunten* aan.

```
class Monster {
  int hitPoints = 100;
}
```

Hiermee is bepaald dat **elk** monster een *hitPoints* heeft. De waarde van dat getal kan per *monster object* verschillen: *Monster 1* kan nog op 100 staan terwijl *monster 2* misschien nog maar 13 over heeft.

De code die in een `class` staat wordt gedeeld met alle `objecten` van die `class` (meestal zeggen we: alle `objecten` van dat `type`, want een `class` is een manier om een `type` te definiëren). Om een `object` van `class` *Monster* aan te maken:

C# of Java:

```
new Monster()
```

(we zeggen dan ook wel dat er gebruik gemaakt wordt van de *new* `operator`)

Hiermee wordt ergens in het geheugen een `object` van `type` *Monster* aangemaakt, we hebben echter géén manier om naar dat `object` te *verwijzen* (*refereren*). Vergelijk het met een ballon met gas: zolang je het touwtje hebt (de

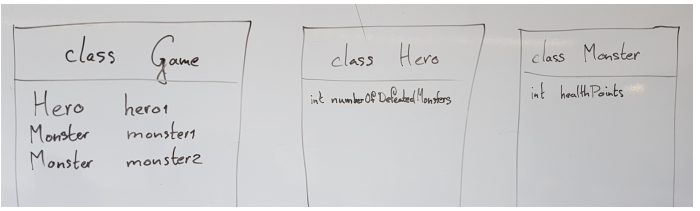

Figure 1.3 Schematische weergave

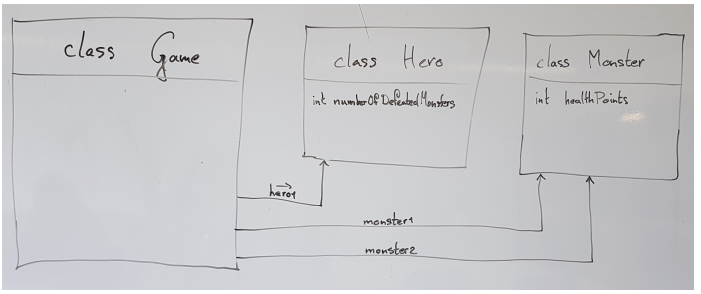

Figure 1.4 Iets handigere schematische weergave

referentie naar de balon) kun je bij de ballon, maar als je het touwtje loslaat kun je niet meer bij de ballon komen.

Zo'n *referentie* kunnen we opslaan in een `Field` (ook wel een variabele genoemd) en dat `Field` moet ergens in een `class` zitten: We maken hiervoor een *Game*-`object` aan dat de referenties naar alle *heroes* en *monsters* bevat. De code van het *Game* `object` komt in de `class` Game te staan.

In `class` Hero is een `Field`

```
int numberDefeatedMonsters = 0;
```

aangemaakt. De held wil namelijk graag dat de hele wereld weet hoeveel monsters er door hem/haar verslagen zijn.

De `class` *Game* heeft referenties naar 1 *hero* en 2 monsters (*monster 1* en *monster 2*) en aan het ervoor vermelde `type` (dat zijn de `class` namen) kun je zien dat de *hero* zich gedraagt zoals in `class` 'Hero' geprogrammeerd is, terwijl de beide monsters zich gedragen volgens de code in `class` Monster.

```
// Fields in de class Game
Hero hero;
Monster monster1;
Monster monster2;


public Game()
```

```
{
   hero = new Hero();
   monster1 = new Monster();
   monster2 = new Monster();
}
```

1.3 Attack

Onze *hero* staat te popelen om een monster te gaan aanvallen. Hiervoor gaan we *gedrag* in de `class` *Hero* programmeren:

Dit wil zeggen dat je naar een `object` van `type` *Hero* een *bericht* kunt sturen die *Attack* heet. Verder vertel je **welk** monster aangevallen wordt en hoeveel *schade* (*damage*) hierbij toegebracht wordt aan het monster (dus hoeveel er van de *hitPoints* punten van het monster af gaan).

Als het bericht *Attack* naar een *Held* `object` gestuurd wordt (in veel programmeertalen praten we niet over *een bericht sturen* maar noemen we het *een methode aanroepen*) wordt de code van die methode uitgevoerd.

De held handelt dit bericht af door een bericht *LooseHitPoints* te sturen naar het monster dat tussen de haakjes genoemd wordt.

Nu gaan we coderen hoe het afhandelen van het bericht er uit kan zien. We zeggen dan dat we de `method` (want zo'n bericht heet in veel programmeertalen een `method`) *Attack* in de `class` *Hero* programmeren

```
void Attack(Monster monster, int damage)
{
  monster.LooseHitPoints(damage);
}
```

ofwel: als een `object` van type *Hero* (want in die class staat deze code) het bericht *Attack* krijgt (met achter *Attack* tussen haakjes aangegeven **welk** monster en hoeveel damage) stuurt ie een bericht *LooseHitPoints* naar het aangegeven monster. De binnengekomen info over hoeveelheid *damage* wordt doorgegeven aan dit nieuwe bericht.

In de methode worden 2 zogenaamde parameters gebruikt, namelijk *monster* van het `type` *Monster* en *damage* van het `type` *int* (tussen haakjes te vinden na de methode naam).

Het woord `void` wil zeggen dat er geen waarde wordt teruggegeven door de methode, Er kan ook in plaats van `void` een zogenaamd `return type` staan dat aangeeft wat voor soort waarde er terug gegeven wordt.

In `class` *Monster* moet vervolgens de `method` *LooseHitPoints* gecodeerd worden:

```
void LooseHitPoints(int damage)
{
```

```
    this.hitPoints = this.hitPoints - damage;
}
```

Uitleg:

- Wederom begint het met *void* omdat de methode niks teruggeeft.
- Dan de methodenaam *LooseHitPoints*.
- Tussen de haakjes de ene parameter, genaamd *damage* en van type *int*.
- Tussen de accolades of *curly brackets* ({ en }) staat een assignment:
  - Een assignment is te herkennen aan het =-teken (spreek uit als **wordt**).
  - Rechts van de = staat een *expressie*, zeg maar een berekening, die uitgerekend (geëvalueerd) wordt. In dit geval: `this.hitPoints - damage`.
  - Het woord *this* geeft aan dat er iets gedaan wordt met het *object* waar we nu 'in' zitten: het specifieke monster dus dat werd aangevallen en dat dus als parameter aan method *Attack* werd meegegeven. In methode *Attack* zie je dat van *DAT* specifieke monster de methode *LooseHitPoints* wordt aangeroepen.
  - *Evaluatie* (berekening) van *this.hitPoints* geeft het *hitPoints*-getal van dat monster.
  - De '`- damage`' zorgt dat de meegegeven waarde van de parameter hier vanaf getrokken wordt.
  - De waarde van *this.hitPoints* (links van de =) wordt de uitkomst van de berekening.

Constructie van een object

Net zoals we bij een methode aanvullende informatie mee kunnen geven in de vorm van `parameters` kunnen we dat bij het aanmaken van een nieuw `object` ook. Hiertoe gebruiken we een `constructor`: Een `constructor` ziet er ongeveer uit als een methode:

```
Monster(int initialHitPoints)
{
  this.hitPoints = initialHitPoints;
}
```

Een `constructor` herken je alsvolgt:

- De constructor lijkt heel erg op een normale methode, maar...

- Er wordt geen `return-type` (of `void`) vermeld.
- De naam (*Monster* in dit geval) is gelijk aan de naam van de `class`.

Constructor in Visual Studio

Je kunt natuurlijk de code hierboven zelf intypen (tussen de accolades van de `class`) maar als je op die plek intypt *ctor* en dan 2x op *tab* drukt doet Visual Studio een deel van het werk voor je.

Als we nu `new Monster(125)` aanroepen vanuit code wordt er een object van type *Monster* geconstrueerd en daarvoor staat na constructie de *hitPoints*-waarde op het meegegeven getal, 125 dus in dit geval.

Bij een `constructor` kunnen (net als bij een *normale* methode) ook meerdere parameters meegegeven worden.

1.4 Wat hebben we nu?

We hebben nu een basis neergezet voor een spel waarin een *hero monsters* kan aanvallen.

Om het werkend te krijgen

Later wordt nog uitgelegd waarom, maar onthoudt vast dat we elk *Field* `private` maken. `Methods` en `classes` mogen `public` zijn.

Code tot nu toe

Voor de volledigheid volgt nu de code van de classes zoals die tot hier beschreven is.

Allereerst de `class` *Game*

```
namespace HereComeTheMonsters
{
    public class Game
    {
        public Game()
        {
            Hero hero = new Hero();
            Monster monster1 = new Monster(125);
            Monster monster2 = new Monster(100);
        }
    }
}
```

dan `class` *Hero*

```
namespace HereComeTheMonsters
{
    public class Hero
    {

        public Hero()
        {
        }


        public void Attack(Monster monster, int damage)
        {
            monster.LooseHitPoints(damage);
        }

    }
}
```

en tot slot `class` *Monster*

```
namespace HereComeTheMonsters
{
    public class Monster
    {

        private int hitPoints = 100;


        public Monster(int initialHitPoints)
        {
            this.hitPoints = initialHitPoints;
        }


        public void LooseHitPoints(int damage)
        {
            this.hitPoints = this.hitPoints - damage;
        }


    }
}
```