Opdracht 3: Betere oplossingen

– Objectgeoriënteerd Programmeren in Greenfoot –

april, 2015

1 Inleiding

In de vorige opgaven maakte je kennis met Greenfoot. Je bent in staat om code te lezen, aan te passen en zelf code te schrijven. Ook kan je generieke oplossingen bedenken en implementeren. In deze opdracht zul je leren om op een slimmere manier code te schrijven. Met deze slimheid is jouw oplossing namelijk efficiënter. Ook kan deze makkelijker hergebruikt worden.

2 Leerdoelen

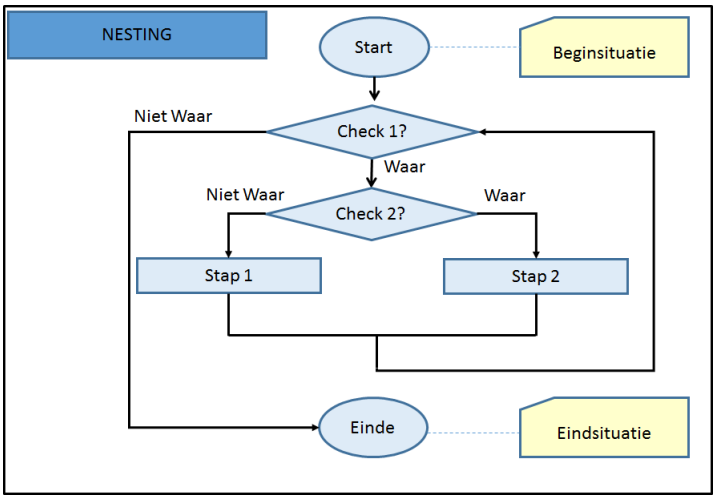
Na het voltooien van deze opdracht kun je:

- nesting in een stroomdiagram en in code herkennen;
- **nesting** toepassen als strategie om een probleem op te lossen;
- stroomdiagrammen **optimaliseren** door deze te vereenvoudigen;
- voorgestelde wijzigingen in een stroomdiagram doorvoeren in de bijbehorende code;
- voordelen noemen van het toepassen van **abstractie**;
- kandidaten voor submethodes in een stroomdiagram herkennen;
- abstractie toepassen in stroomdiagrammen en bijbehorende code door gebruik te maken van **submethodes**;
- beschrijven wat het verband is tussen, aan de ene kant, het ontwerpen van een programma (met bijvoorbeeld een stroomdiagram) en gestructureerd werken en aan de andere kant het aantal implementatiefouten;
- beoordelen in hoeverre een oplossing **generiek** is;
- in eigen woorden uitleggen hoe de *Run* in Greenfoot werkt (als herhaling van *Act*);
- een Greenfoot programma stoppen;
- eigen programmacode opstellen, compileren, uitvoeren en testen.

3 Uitleg

Nesting
De constructies opeenvolging, keuze en herhaling kunnen ook in verschillende combinaties met elkaar gebruikt worden. Ze kunnen achter elkaar gebruikt worden, of in elkaar. Dat laatste heet *nesting*.

Stroomdiagram:
Zie hier een voorbeeld van een stroomdiagram van een **`if`** .. `then` .. **`else`** genest in een **`while`**.



Figuur 1: Stroomdiagram van een keuze genest in een herhaling

Toelichting stroomdiagram:

- Eerst wordt er in de eerste ruit controleert of de conditie 'Check 1?' 'waar'.
- Als de conditie 'Niet waar' is, dan is de methode afgelopen.
- Als de conditie 'Waar' is, dan wordt de conditie in de tweede ruit 'Check 2?' gecontroleerd.
  - Als de tweede conditie 'Check 2?' 'Niet waar' is, dan wordt 'Stap1' uitgevoerd.
  - Als de tweede conditie 'Check 2?' 'Waar' is, dan wordt 'Stap2' uitgevoerd.
- Daarna wordt er in beide gevallen teruggegaan naar de eerste ruit (controleren of conditie 'Check 1?' 'Waar' is). Is conditie 'Check 1?' nog steeds 'Waar'? Dan wordt de pad naar 'Check 2?' weer vervolgd, enz (dit is een loop). Anders is de methode afgelopen.

Code:
In de code ziet dat er zo uit:

```
void methodeNaam( ) {            // methode met een herhaling
                                 // herhaling
      while ( check1 ( ) ) {     // check de conditie in de ruit
                                 // als de conditie waar is
           if ( check2 ( ) ) {   // check ook de conditie in de 2e ruit
                                 // als de conditie in de 2e ruit ook waar is
                stap2 ( );        // roep de methode aan in de rechthoek

           } else {              // als de conditie in de 2e ruit niet waar is
                stap1 ( );       // roep de methode aan in de rechthoek
           }
      }
}
```

Toelichting code:

- Eerst wordt de conditie `check1( )` gecontroleerd.
- Als de conditie van de **`while`** gelijk is aan **`false`** (dus `check1( ) ==` **`false`**), dan is de methode afgelopen.

- Als de conditie van de **`while`** gelijk is aan **`true`** (dus `check1( ) == true`), dan wordt de conditie achter de **`if`** gecontroleerd (dus `check2( )`).
  - Als de conditie achter de **`if`** gelijk is aan **`true`** (dus `check2( ) == true`), dan wordt de code tussen de accolades { en } uitgevoerd (dus `stap2( )`).
  - Als de conditie achter de **`if`** gelijk is aan **`false`** (dus `check2( ) == false`), dan wordt de code achter de **`else`** tussen de accolades { en } uitgevoerd (dus `stap1( )`).
- In beide gevallen wordt er teruggegaan naar de controle van de **`while`** conditie (dus `check1( )`). Als deze nog steeds **`true`** is, dan wordt de code tussen de accolades weer uitgevoerd (loop). Anders is de methode afgelopen.

Toevoeging: de opeenvolgingen, herhalingen en keuzes kunnen in willekeurige volgorde voorkomen. Ook kunnen ze als onderdeel binnen de andere voorkomen.

4 Opgaves

4.1 Optimaliseren

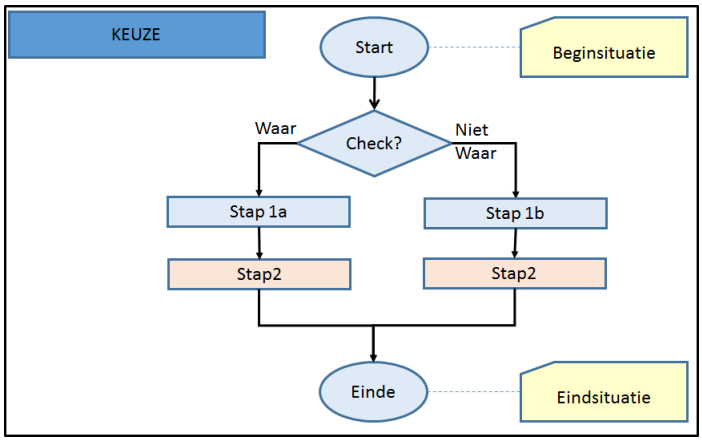
Optimaliseren
Een stroomdiagram, en de bijbehorende code, kun je soms eenvoudiger en overzichtelijker maken zonder dat dat gevolgen heeft voor de werking van het programma of de eindsituatie. Zo'n vereenvoudiging noem je een *optimalisatie*. Een optimalisatie heeft geen gevolgen voor de werking van het programma of eindsituatie.

Door een optimalisatie:

- wordt het diagram en de code overzichtelijker en vaak beter leesbaar;
- zijn minder regels code nodig;
- wordt zowel het diagram als de code makkelijker te onderhouden (d.w.z. aan te passen).

Dit verkleint de kans op fouten.

Doubleurs
Activiteiten die onnodig dubbel voorkomen
Voorbeeld:

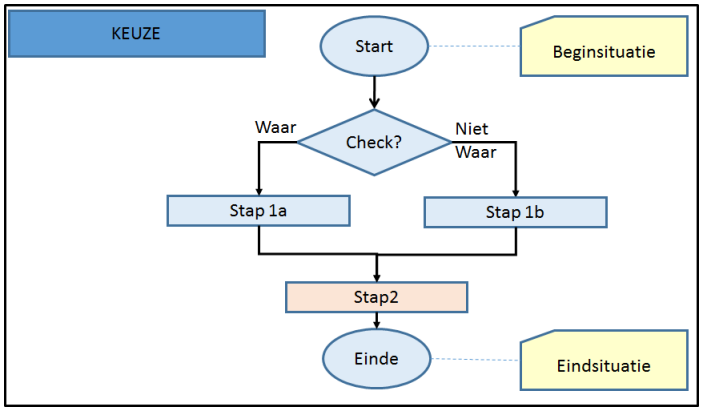


Figuur 2: Stroomdiagram met dubbele activiteiten

In het stroomdiagram in figuur 2 komt 'Stap2' twee keer voor.

- Als de conditie 'Waar' is, wordt 'Sstap 1a' uitgevoerd en daarna 'Stap2';
- Als de conditie 'Niet waar' is, wordt 'Stap 1b' uitgevoerd en daarna 'Stap2';
- Daarna is de methode afgelopen.

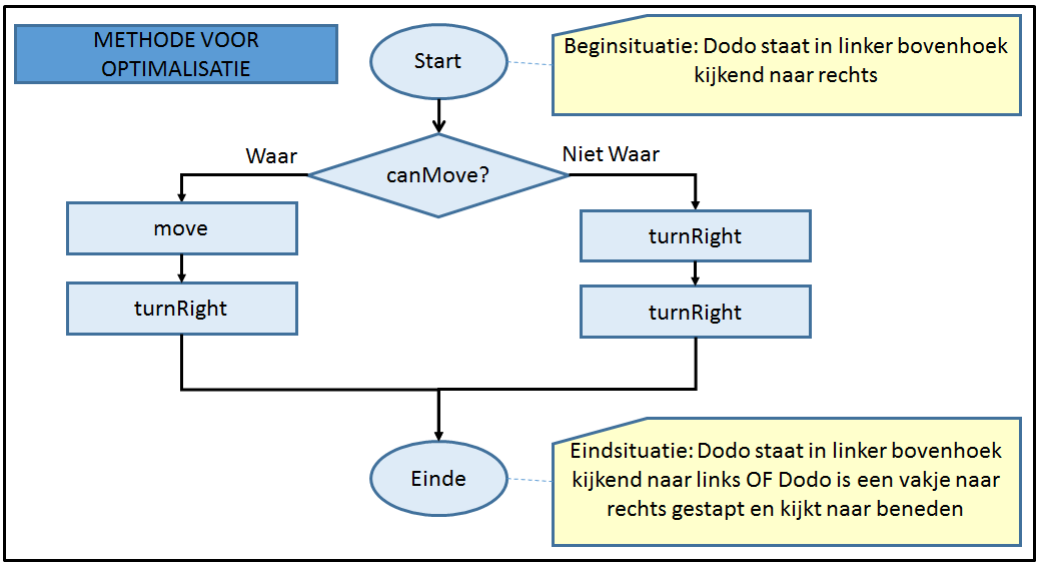
In beide gevallen wordt 'Stap2' als laatste uitgevoerd. Dit stroomdiagram kan overzichtelijker gemaakt worden door deze twee stappen samen te nemen. Er staat nu nog maar één aanroep van 'Stap2'. Deze aanpassing heeft geen gevolgen voor de werking van het programma of eindsituatie. Zie figuur 3 voor het stroomdiagram na deze optimalisatie.



Figuur 3: Stroomdiagram zonder dubbele activiteiten

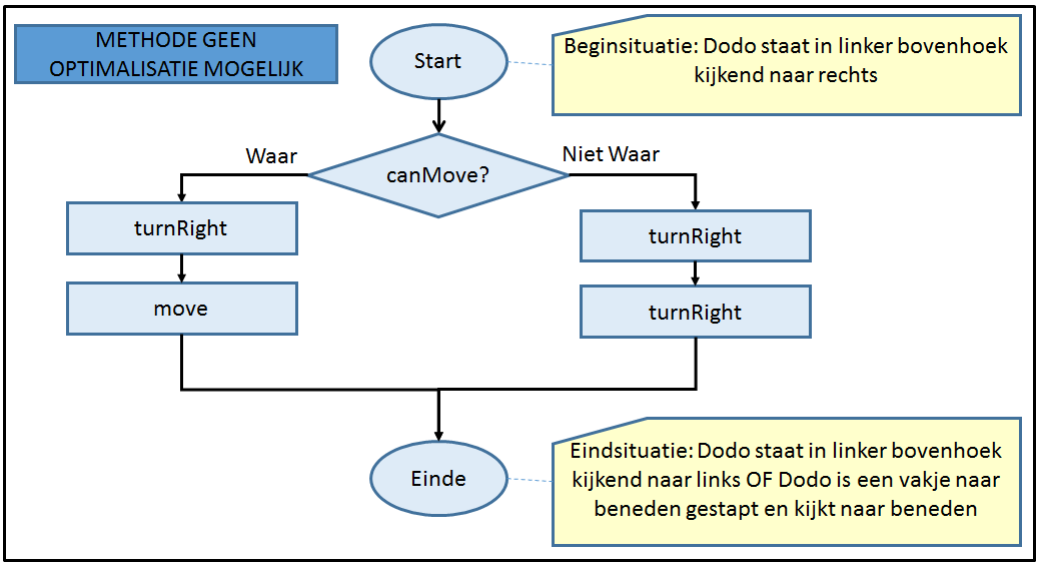
4.1.1 Optimaliseren doubleurs

In de volgende opgaven ga je een stroomdiagram optimaliseren.



Figuur 4: Stroomdiagram vóór optimalisatie

1. Bekijk het stroomdiagram in figuur 4.
2. Vergelijk de stappen in het linker pad met de stappen in het rechter pad. Wat valt je op?
3. Optimaliseer het stroomdiagram. Let er op dat deze aanpassing geen gevolgen voor de werking van het programma of eindsituatie heeft.
4. Bekijk het stroomdiagram in figuur 5. Leg uit waarom hier niet een vergelijkbare optimalisatie gedaan kan worden.



Figuur 5: Stroomdiagram waarbij optimalisatie niet mogelijk is

4.2 Submethodes

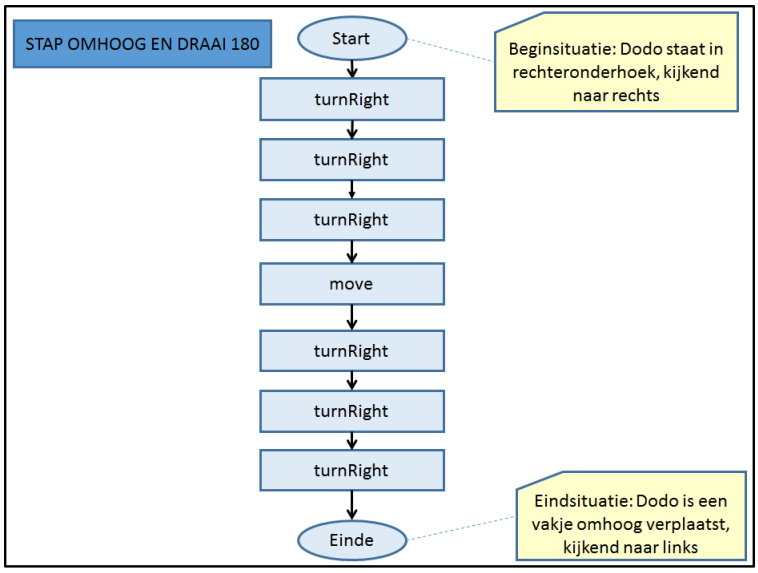
Abstractie
Een stroomdiagram, en de bijbehorende code, kun je overzichtelijker maken door een reeks van stappen apart te benoemen. Je maakt voor die stappen een *submethode* met een eigen substroomdiagram. Je verwijst dan vanuit jouw eerste diagram naar het subdiagram. In de code doe je hetzelfde: vanuit de methode verwijs je naar de submethode. Dit proces heet *abstractie*.

Toelichting:
Abstractie pas je toe bij:

- herhaling: als een bepaalde reeks van instructies vaker voorkomt;
- veel stappen: als het diagram door de vele stapjes (bijvoorbeeld: meer dan zeven) onoverzichtelijk wordt.

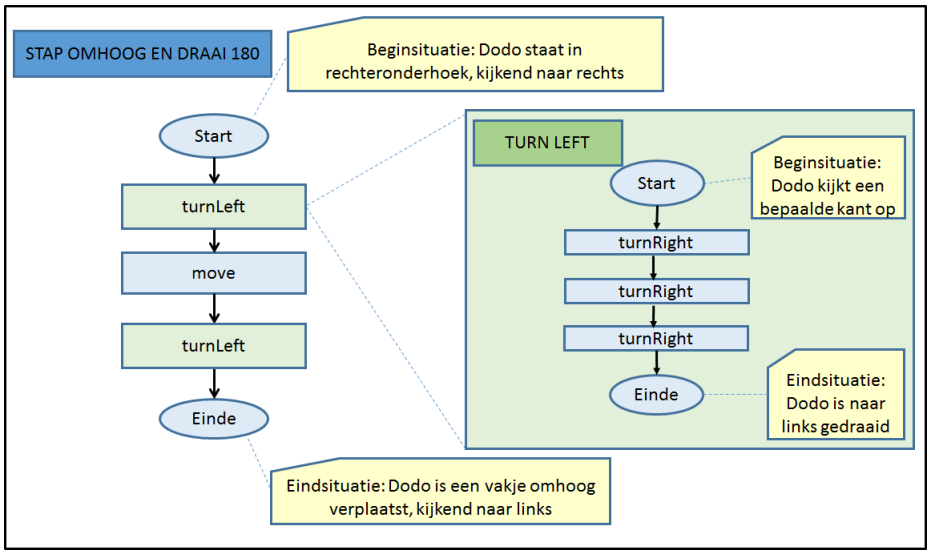
Door abstractie wordt de programmacode makkelijker te begrijpen, aan te passen en uit te breiden. Ook het testen wordt makkelijker, want de submethodes kunnen los van elkaar getest worden. Een fout (en de oorzaak daarvan) heb je zo sneller gevonden. Daarna wordt het geheel nog getest. Ook is het minder foutgevoelig doordat sommige stukken code niet nodeloos worden herhaald. De submethode kan ook hergebruikt worden in andere methodes, of zelfs (door andere programmeurs) voor andere programma's.

Voorbeeld:



Figuur 6: Stroomdiagram vóór optimalisatie

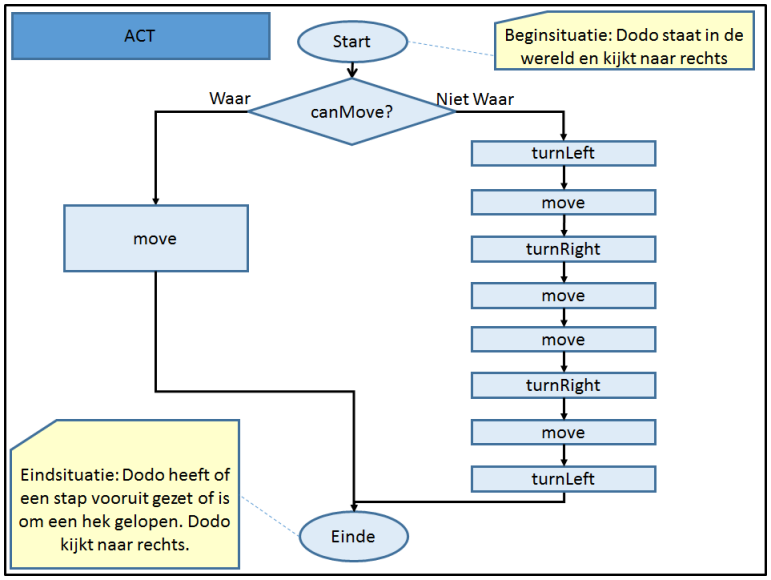
In het stroomdiagram in figuur 6 wordt een aantal stappen herhaald. Dit stroomdiagram kan overzichtelijker gemaakt worden door voor deze stappen een aparte submethode (en subdiagram) te maken. In de oorspronkelijke methode roepen we de nieuwe methode aan. Deze aanpassing heeft geen gevolgen voor de werking van het programma of eindsituatie. Zie figuur 7 voor het stroomdiagram na de optimalisatie.



Figuur 7: Stroomdiagram na optimalisatie

4.2.1 Submethodes gebruiken

Nu gaan we een stroomdiagram uit opdracht 2 'Om een hek heen lopen' overzichtelijker maken. Hetzelfde doen we ook voor de bijbehorende code.



Figuur 8: Stroomdiagram vóór optimalisatie

Zoals bij elke aanpassing, doen we dat stapsgewijs. Dat verkleint de kans op fouten. We volgen daarvoor het stappenplan voor het aanpassen van code zoals beschreven in opdracht 2. Dit doen we de eerste keer samen:

1. Bekijk het stroomdiagram hierboven. Noem de reeks op van opeenvolgende stappen (aan de 'Niet Waar' kant). Welk van de **Stroomdiagram regels** beschreven in opdracht 2 wordt overtreden?

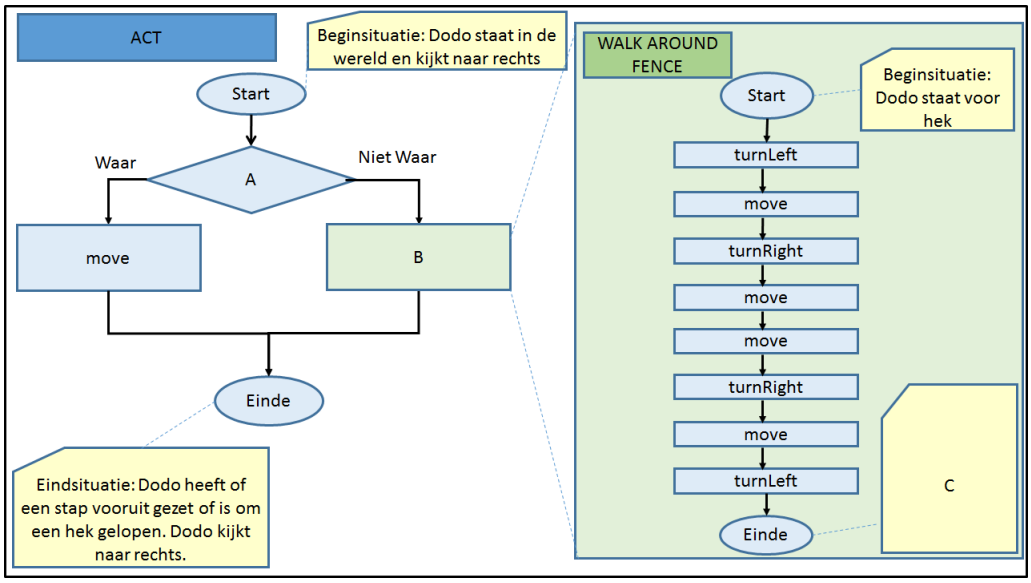
2. We maken het stroomdiagram overzichtelijker door een submethode te maken. Dit doen we op dezelfde wijze als in het theorieblok hierboven:

   (a) Voor de reeks opeenvolgende stappen die je zojuist opgenoemd hebt, maken we een submethode genaamd: 'walkAroundFence'.

   (b) Deze zetten we in een tweede (nieuw) subdiagram met dezelfde naam 'walkAround-Fence'.

   (c) Vanuit het eerste diagram verwijzen we naar het nieuwe subdiagram (met een blauwe stippellijn).

   (d) Het resultaat zie je in figuur 9. Het nieuwe subdiagram zie je in het groen.



Figuur 9: Stroomdiagram `act( )` met aanroep van een submethode

3. Wat zijn A, B en C in het diagram hierboven?

4. Nu passen we de code aan zodat deze overeenkomt met het nieuwe stroomdiagram.

   (a) Open jouw scenario uit opdracht 2. Je gaat dus verder met jouw eigen code. Is dat écht onmogelijk, dan mag je gebruik maken van het scenario **'MadagaskarOpdr3'**.

   (b) Open de wereld "world_eggFenceInWay". Gebruik hiervoor **`void`** `populateFromFile( )`.

   (c) Open de code voor `MyDodo` in de editor.

   (d) We maken een nieuwe submethode voor MyDodo **`void`** `walkAroundFence( )` die overeenkomt met het subdiagram 'walkAroundFence'. Tik het volgende over:

```
/**
*  Deze methode zorgt ervoor dat Mimi ...
*/
public void walkAroundFence () {


}
```

   (e) Zet tussen de accolades { en } alle methode aanroepen die in het subdiagram 'Walk Around Fence' staan. Dat zijn precies dezelfde stappen die je in opdracht 2 opgave 'Om een hek heen lopen' bedacht hebt.

(f) Voeg boven de submethode commentaar toe om aan te geven wat deze doet.

(g) Compileer de code.

(h) Zet Mimi ergens midden in de wereld neer, met een hekje recht voor haar.

(i) Wat verwacht je dat Mimi doet als je **void** `walkAroundFence()` uit zou voeren?

(j) Test de methode door met de rechtermuisknop op haar te klikken en **void** `walkAroundFence()` te kiezen. Doet Mimi precies wat in jouw subdiagram beschreven staat?

(k) Haal het hekje vóór Mimi weg.

(l) Wat verwacht je dat Mimi doet als je nu **void** `walkAroundFence()` uit zou voeren?

(m) Test de methode door met de rechtermuisknop op haar te klikken en **void** `walkAroundFence()` te kiezen. Doet Mimi precies wat in jouw subdiagram beschreven staat?

(n) We passen nu de `act`-methode aan zodat deze de nieuwe submethode `walkAroundFence` alléén aanroept als Mimi niet rechtdoor kan lopen. Pas de code in **void** `act( )` aan zodat in de **else** deze nieuwe methode wordt aangeroepen, dus:

```
public void act( ){
    if ( canMove( ) ){
        move( );
    } else {
        walkAroundFence( );
    }
}
```

(o) Compileer, run en test het programma met *Act*. Door op *Act* te klikken test je nu niet alleen `walkAroundFence`, maar ook de voorwaarde dat ze dat alléén doet als ze voor een hekje staat. Test dus het geval met een hekje voor Mimi en het geval zonder hekje. Werkt het programma niet correct? Volg de stappen zoals beschreven bij hoofdstuk 'Debuggen' van opdracht 2.

(p) Wat gebeurt er als je **void** `act( )` meerdere keren achter elkaar aanroept? Controleer dit door op de *Run*-knop de drukken.

5. Sla jouw scenario op. We gaan hier straks (in opgave 4.4) mee verder.

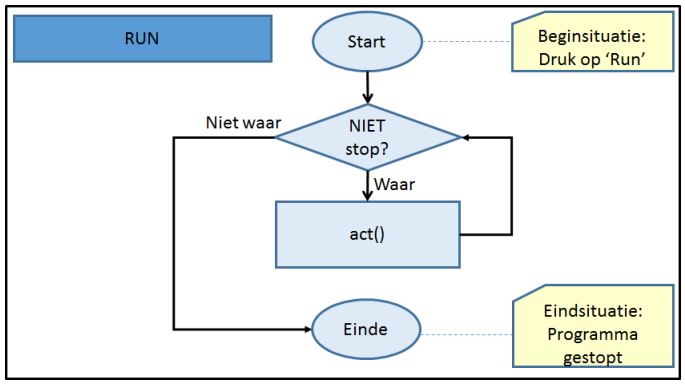
   (a) Kies in Greenfoot 'Scenario' in het bovenste menu, en dan 'Save As ...'.

   (b) Vul de bestandsnaam aan met jouw eigen naam en het opgavenummer 3a, bijvoorbeeld `Opdr3a_Michel`.

4.3 Run

Run
Als je in Greenfoot op de *Run*-knop drukt dan wordt het gehele scenario uitgevoerd. Dat betekend dat voor elke actor uit het scenario steeds opnieuw de `act`-methode wordt aangeroepen. Dit stopt pas als je op de 'Pauze'-knop drukt. Het stopt ook als er ergens in de code `Greenfoot.stop( )` is aangeroepen.

Stroomdiagram: Het stroomdiagram in figuur 10 geeft het gedrag van de *Run* weer.



Figuur 10: Stroomdiagram van *Run*

Toelichting stroomdiagram: De gebruiker drukt op *Run*.

- Eerst wordt gecontroleerd of de conditie in de ruit ('NIET stop?') 'Waar' is;
- Als de conditie 'Niet waar' is (de gebruiker heeft op pauze gedrukt of er is ergens `Greenfoot.stop( )` aangeroepen), dan is de *Run*-methode afgelopen.
- Als de conditie 'Waar' is, wordt `act` uitgevoerd.
  Daarna wordt er teruggegaan naar de ruit en wordt de conditie opnieuw gecontroleerd. Is de conditie 'NIET stop' nog steeds 'Waar'? Dan wordt het pad van 'Waar' weer vervolgd, net zolang tot totdat de conditie 'Niet waar' wordt (dit heet een loop). Anders is de methode afgelopen.

Toelichting: Voor het uitvoeren van een methode kun je dat op één van de volgende drie verschillende manieren doen:

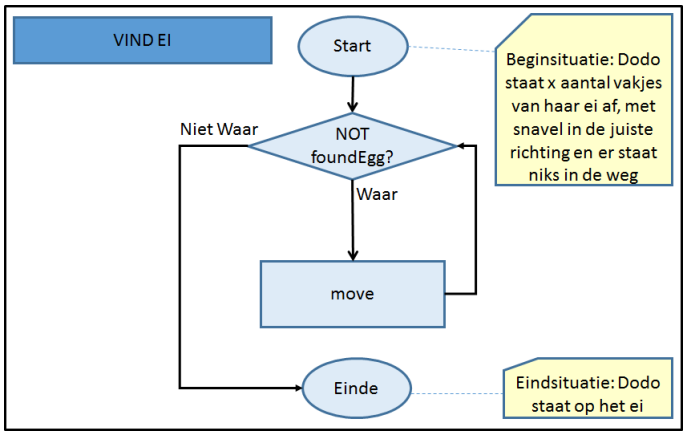
- De methode direct aanroepen. Dit doe je door met je rechtermuisknop op Mimi te klikken en deze methode te selecteren.
- Op de *Act*-knop drukken. Dan moet de methode wel in **`void`** `act( )` aangeroepen zijn.
- Op de *Run*-knop drukken. Ook geldt hier dat de methode vanuit **`void`** `act( )` aangeroepen moet zijn. De methode wordt niet één keer, maar meerdere keren aangeroepen. Dat kan handig zijn als je niet één geval maar meerdere gevallen achter elkaar wilt uitproberen. Bijvoorbeeld heel veel stapjes zetten om een ei te vinden.

Bekijk het stroomdiagram in figuur 10 hierboven.

1. Welke constructie hoort bij dit stroomdiagram?
2. Wat moet er gebeuren voordat het pad 'Niet waar' gevolgd wordt?
3. Teken een nieuw stroomdiagram waarbij je 'NIET stop' vervangt door 'stop' en 'Niet waar' en 'Waar' verwisselt. Is het stroomdiagram nog correct? Leg uit waarom niet.

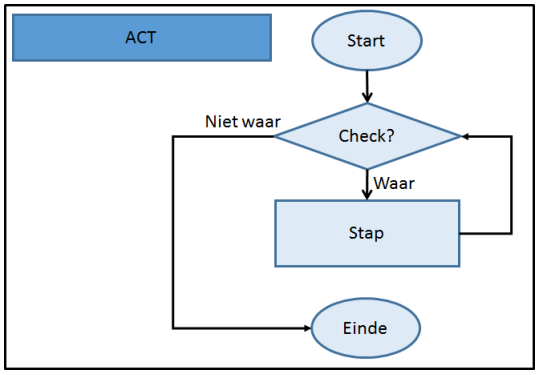
Ingebouwd in Greenfoot: `while`-loop in de *Run*
De stroomdiagrammen van de **`while`**-loop in een *Run* zijn steeds hetzelfde. In een aantal voorbeelden heb je algoritmes uitgewerkt waarin een **`while`**-loop voorkomt om bepaalde

stappen te herhalen. Bijvoorbeeld in het 'vind-het-ei'-algoritme (opdracht 2, onderdeel 4.3.1). Het bijbehorende stroomdiagram zie je in figuur 11 hieronder.

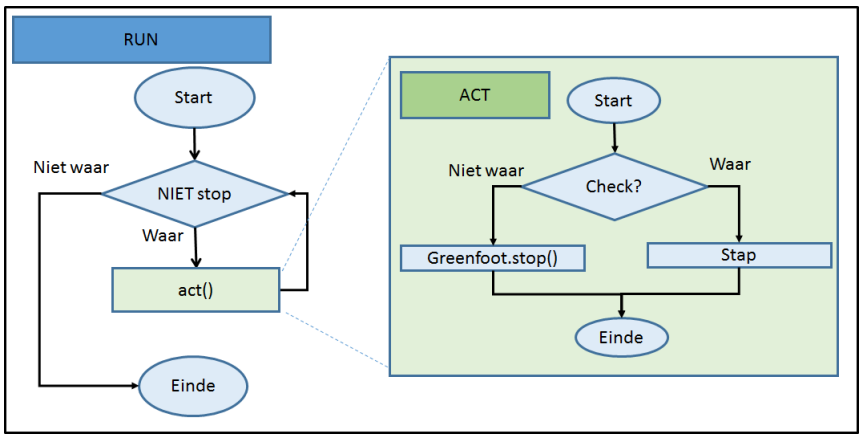


Figuur 11: Stroomdiagram voor 'vind-het-ei' (opdracht 2, onderdeel 4.3.1)

De *Run* zorgt ervoor dat `act` steeds opnieuw wordt aangeroepen. Dat hoort bij Greenfoot. Maar, in dit geval, is na de eerste uitvoering van `act` de algoritme klaar. Iedere volgende aanroep van `act` heeft geen effect meer. Dit maakt de *Run* zinloos. *Run* is eigenlijk alleen maar zinvol als je algoritme bestaat uit een herhaling van steeds weer dezelfde stappen. Deze stap plaats je dan in de `act`-methode. Dus in plaats van het volgende stroomdiagram:



kun je beter het volgende stroomdiagram tekenen:



In de stroomdiagrammen zijn de **while**-loop van *Run* zijn steeds hetzelfde. Daarom laten we deze **while**-loop vanaf nu steeds weg. De loop vind je ook nergens terug in je code: hij zit in Greenfoot ingebouwd.

In opdracht 2, onderdeel 4.3.1 heb je het 'vind-het-ei'-algoritme geïmplementeerd.

1. Beschrijf het doel van het algoritme.
2. Bekijk het stroomdiagram van **void** `act( )` dat je daarbij gebruikt hebt. Kun je die niet terugvinden? Kijk dan naar figuur 11 in het bovenstaande theorieblok.
3. Welke constructie hoort bij dit stroomdiagram?
4. Welke afspraak hebben we nu gemaakt over zo'n constructie in de `act`?
5. Teken het stroomdiagram voor het 'zoek-het-ei'-algoritme zonder de **while** te gebruiken.
6. We gaan testen of het programma zonder **while** inderdaad doet wat we verwachten. Daarvoor moeten we de code aanpassen:
   (a) Vervang de code in de `act` door de code dat bij het oorspronkelijke stroomdiagram hoort (met de **while**). Dat is de volgende code:

```
/**
 * Zet steeds stappen in de kijkrichting totdat ei gevonden is
 */
public void act() {
    while( !foundEgg() ){
        move();
    }
}
```

   (b) Pas de code aan zodat deze overeenkomt met het nieuwe stroomdiagram.
   (c) Open de wereld: 'world_Aanroepen6movesAlsWhile.txt'.
   (d) Test jouw algoritme door op de *Run*-knop te drukken. Doet het programma wat je verwacht?
   (e) Wat kan je zeggen over de afspraak dat je in onderdeel 4 hebt beschreven?

We hebben nu gezien dat een **while** in de `act`-methode vervangen kan worden door een **if**-statement. We hebben afgesproken dat we liever de **if**-statement gebruiken.

Run of Act?
"Waarom heeft men eigenlijk deze *Run*-knop toegevoegd?", vraag je je wellicht af. Daarvoor zijn verschillende redenen:

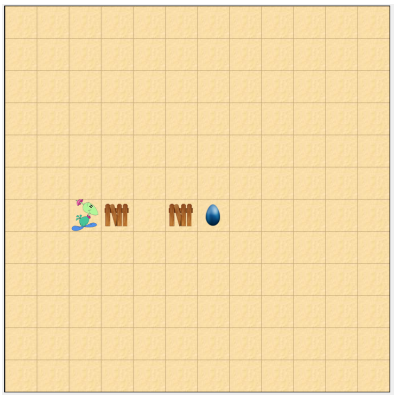
- Er zijn scenario's waarin meerdere actoren voorkomen met een `act`-methode. Deze situatie krijg je al als je in een van je scenario's twee of meer `MyDodo`-instanties plaatst. Probeer het maar eens uit. Wat je dan meestal wil is dat al deze objecten (min of meer) gelijktijdig iets doen. Dat krijg je niet voor elkaar als je de complete taak in de **void** `act()` hebt staan, maar wel als je de taak hebt opsplitst in kleine stappen.

- Wellicht wil je je programma laten reageren op *gebruikersinvoer* (muisklikken of toetsen die worden ingedrukt). Met de *Run*-functionaliteit ben je niet gedwongen om alles in één keer in je `act` te doen. Doe je dat wel, dan kan het soms lang duren voordat het programma op de gebruiker reageert. Dat is niet de bedoeling, en voor de gebruiker erg frustrerend. Door gebruik te maken van de *Run* kan jouw programma snel reageren op de gebruikersinvoer.

Run als het kan
Vanaf nu stellen we de programma's zo op dat in de `act`-methode steeds één stap van het algoritme gezet wordt. Als je dan op de *Run*-knop drukt wordt het programma als geheel uitgevoerd. Voor de meeste programma's blijkt dat geen echt probleem te zijn. Pas als je de actoren iets laat doen wat behoorlijk complex is, dan kan dat lastiger worden.

4.4 Ei vinden met meerdere obstructies

We gaan nu verder met het scenario dat je had opgeslagen in opgave 4.2.1 onderdeel 5. Bekijk nu de volgende wereld:



Figuur 12: Scenario voor opgave 4.4

1. Open het scenario dat je had opgeslagen met bestandsnaam jouw eigen naam en het opgavenummer 3a.
2. Open de wereld: 'world_egg2FenceInWay'.
   (a) Klik met de rechtermuisknop in de wereld.
   (b) Kies **`void`** `populateFromFile( )`.
   (c) Ga naar het map 'worlds'.
   (d) Kies 'world_egg2FenceInWay.txt'
3. Is jouw oorspronkelijke code generiek genoeg om ook in deze wereld te werken?
4. Test de code met de *Act* knop door hier 2 keer op te klikken. Loopt Mimi keurig om beide hekjes heen? Zo niet, dan moet je aanpassingen maken.
5. Open opnieuw de wereld: 'world_egg2FenceInWay'.
6. Wat gebeurt er nu als je op *Run* klikt?
7. Omschrijf in jouw eigen woorden waarom Mimi niet stopt na het vinden van het ei (tip: lees eventueel opnieuw de uitleg in hoofdstuk 4.3).

We hebben nu gezien dat door op *Run* te klikken, de code in de *Act* herhaaldelijk aangeroepen wordt.

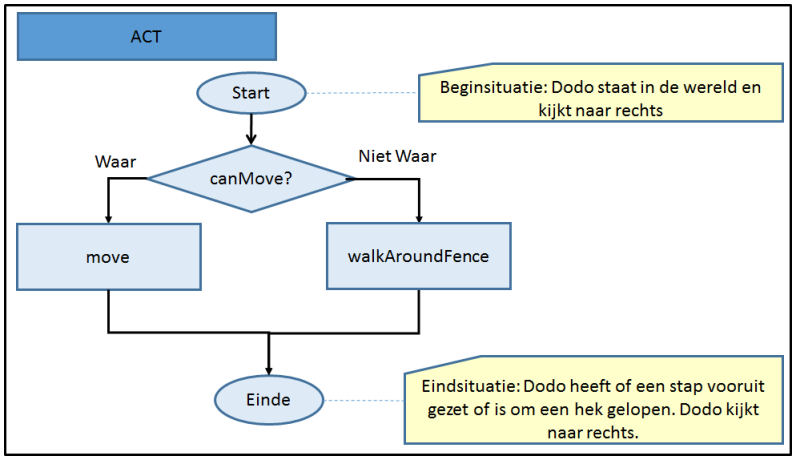
4.5 Complimentje geven

`String`
Een stuk tekst heet in Java `String`. `String` is een *type*, net als **`int`** en **`boolean`**. Een parameter kan ook van het type `String` zijn en dus kan je bij aanroep van een methode een tekst worden meegegeven. Als je een letterlijk stukje tekst in je programma wil gebruiken dan moet je dit tussen aanhalingstekens plaatsen. Verder kun je met teksten allerlei dingen doen zoals bijvoorbeeld twee teksten aan elkaar knopen van met behulp van '+'. Zo levert `"Hallo"+ " Mimi"` de tekst `"Hallo Mimi"` op.
Voorbeeld: De methode `showCompliment(String complimentje)` toont een dialoog (pop-up venster) met daarin een stuk tekst. De methode krijgt als parameter het 'complimentje' van het type `String` mee.

Figuur 13: Resultaat na aanroep van:`showCompliment( "Gefeliciteerd!");`

We gaan nu de `act( )` uitbreiden. Als Mimi haar ei vindt geven we haar een complimentje.

1. Als uitgangspunt nemen we het stroomdiagram die je hebt aangevuld in opgave 4.2.1.



Figuur 14: Stroomdiagram voor `act` voor "loop om een hek"

2. Als Mimi het ei vindt moet er een compliment getoond worden. Breid het diagram hiermee uit.
3. Om het compliment te tonen kun je gebruik maken van de methode `showCompliment( String complimentje )`.
4. Pas de code van `act( )` aan zodat het compliment getoond wordt.
5. Pas het commentaar boven de `act( )` methode ook aan.
6. Compileer, run en test het programma. Werkt het zoals je verwacht?

7. Verander de tekst van het compliment zodat duidelijk is waarom je Mimi feliciteert.

8. Compileer en test het programma opnieuw. Werkt het zoals je verwacht?

We hebben nu gezien hoe je na beide takken van een **`if`**-statement (zowel de **`if`** als de **`else`**) een andere methode kunt aanroepen. Ook hebben we gezien hoe je een dialoog met een bepaalde tekst kunt tonen.

4.6 Programma stoppen

Als je nu op *Run* klikt zie je dat Mimi niet stopt na het vinden van de ei. Dat komt door de 'verborgen '**`while`** loop die in de *Run*. Kijk eventueel terug bij opgave 4.3. We gaan nu het programma aanpassen zodat deze stopt zodra de ei gevonden is:

1. Open de code van `MyDodo` en ga naar de methode `act( )`.

2. Voeg code toe om de programma te stoppen zodra een ei gevonden is. Tip: gebruik `Greenfoot.stop( );`.

3. Compileer en test het programma met *Run*. Werk het zoals je verwacht?

We hebben nu gezien hoe je de *Run* van een Greenfoot programma kunt onderbreken om het programma te stoppen.

4.7 Programma testen in meerdere werelden

We hebben nu inmiddels aardig wat aanpassingen gemaakt in de code. Het is tijd om even terug te blikken. Wat kan Mimi inmiddels goed, en wat nog niet.

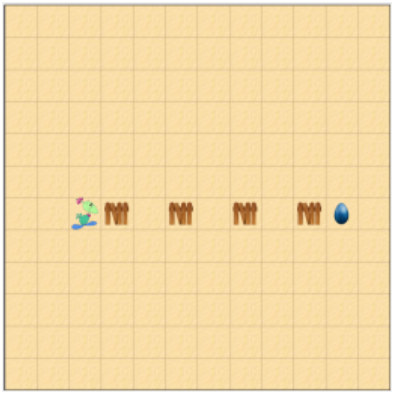
1. Wat is het doel van jouw programma? Beschrijf wanneer je vindt dat het correct werkt. Welke eindsituatie hoort daarbij?

2. Open de wereld 'world_egg3FenceInWayWithSpace'.

3. Test het programma met *Run*.

4. Verklaar waarom Mimi ook in deze wereld het ei weet te vinden.

5. Is jouw code generiek genoeg om ook in de volgende wereld te werken?



Figuur 15: Scenario met vier hekjes

6. Wat kan je zeggen over hoe generiek jouw programma is?

7. Sleep Mimi, de hekken en het ei naar verschillende plekken in de wereld. Test of het programma correct werkt. Probeer ook een aantal andere opstellingen.
8. In welke beginsituaties werkt het programma niet correct?
9. Beschrijf het beginsituatie waarin het programma wel correct werkt.
10. Geef minstens twee voorstellen voor verbeteringen van jouw programma aan (deze hoef je niet uit te werken, alleen benoemen).
11. Wat zijn voor jou de twee belangrijkste dingen die je geleerd hebt in deze opdracht?

We hebben nu gezien wat dat een generiek programma in vele verschillende situaties werkt.

5 Samenvatting

Je hebt geleerd:

- hoe de *Act* en de *Run* werken;
- combinaties van opeenvolgingen, keuzes en herhalingen toe te passen in stroomdiagrammen en in code;
- stroomdiagrammen en code te optimaliseren;
- gestructureerd en stapsgewijs code aan te passen en te testen;
- na te gaan of een oplossing correct.

6 Jouw werk opslaan

Je bent klaar met de derde opdracht. Sla je werk op, want je hebt het nodig voor de volgende opdrachten.

1. Kies in Greenfoot 'Scenario' in het bovenste menu, en dan 'Save As ...'.
2. Vul de bestandsnaam aan met jouw eigen naam en het, bijvoorbeeld: `Opdr3_Michel`.

Alle onderdelen van het scenario bevinden zich nu in een map die dezelfde naam heeft als de naam die je hebt gekozen bij 'Save As ...'.