Opdracht 4: Overzichtelijker en generieker

– Algoritmisch Denken en Gestructureerd Programmeren in Greenfoot –

©2015 Renske Smetsers-Weeda & Sjaak Smetsers

Op dit werk is een creative commons licentie van toepassing.

https://creativecommons.org/licenses/by/4.0/

1 Inleiding

In de vorige opgaven heb je zelf oplossingen bedacht en geïmplementeerd. Ook heb je geleerd om generieke oplossingen te maken. In deze opdracht laat je Mimi ingewikkeldere dingen doen. Dit doe je door de dingen die je tot nu toe geleerd hebt slim te combineren.

2 Leerdoelen

Na het voltooien van deze opdracht kun je:

- gebruik maken van de **logische operatoren** `!`, && en || ;
- een eigen `boolean` methode opstellen;
- **condities** opstellen met combinaties van logische operatoren en `boolean` methodes;
- toepassen van **geneste** `if` .. `then` .. `else` statements;
- `return`-statements gebruiken in stroomdiagrammen en code;
- complexe algoritmes ontwerpen met gebruik van stroomdiagrammen;
- toepassen van methodeaanroepen en resultaattypen;
- toepassen van naamgevingsafspraken voor methoden;
- **modularisatie** toepassen: submethodes afzonderlijk ontwerpen, schrijven, testen en vanuit de code aanroepen;
- beoordelen of begin- en eindsituaties aan elkaar gelijk zijn in een accessormethode;
- **kwaliteitseisen** van programma's en programmacode benoemen;
- beschrijven wat het verband is tussen implementatiefouten en gestructureerd werken en modularisatie;
- beoordelen of een programma/code voldoet aan bepaalde kwaliteitseisen;
- fouten in de code opsporen en analyseren, foutmeldingen interpreteren en aan de hand hiervan problemen herstellen (debuggen).

3 Instructies

Bij deze opdracht ga je verder met jouw code uit opdracht 3. Je hebt dus het scenario nodig dat je na opdracht 3 hebt opgeslagen `Opdr3_jouwNaam`.

4 Uitleg

Logische operatoren
Java heeft drie logische operatoren. Je hebt er al een aantal van gezien. Deze kun je combineren met **`boolean`** methodes om condities te beschrijven.

Operator	Betekenis	Voorbeeld
&&	EN	`facingNorth()` && `fenceAhead()`
||	OF	`fenceAhead()` || `borderAhead()`
!	NIET	`!fenceAhead()`

5 Opgaven

In de volgende opgaven gaan we Mimi programmeren om haar allerlei dingen te laten doen. Het is daarbij belangrijk om Mimi zo slim mogelijk te maken. We leren haar nieuwe dingen, maar willen dat ze die dan ook in alle vergelijkbare gevallen kan. We zijn dus op zoek naar *generieke oplossingen*.

Bijvoorbeeld, als Mimi in het bovenstaande scenario haar ei moet vinden, dan zeggen we niet: "zet 3 stappen". Dat werkt namelijk niet als haar eitje één vak verderop ligt. We zeggen dan liever: "Zolang niet ei gevonden, zet een stap".

5.1 Eigen functionaliteit: aparte methode

Modularisatie
Tot nu toe heb je steeds code in de `act`-methode geschreven. Breid je de code steeds verder uit? Dan wordt het op een gegeven moment onoverzichtelijk. Een ander nadeel is dat je telkens bij elke nieuwe opgave jouw code vervangt door andere code. De 'oude' code heb je dan niet meer.

Beter is het om steeds per deelprobleem (of opgave) een aparte methode te schrijven. Deze kun je dan ook apart testen. Dit heet modularisatie.

Stappenplan Modularisatie:

1. Bedenk het algoritme.
2. Teken het bijbehorende stroomschema.
3. Schrijf de bijbehorende code. Houd je aan de naamgevingsafspraken (zie opdracht 1 'Naamgevingsafspraken').
4. De methode test je afzonderlijk door met de rechtermuisknop op het object in de wereld te klikken en de methode te kiezen. Werkt de methode niet correct? Volg de stappen in opdracht 2 'Debuggen'. Pas als je zeker weet dat jouw methode werkt zoals verwacht wordt, ga je door.

5. Pas de `act( )` aan zodat deze de nieuwe methode aanroept. De methode kan eventueel ook vanuit andere methoden aangeroepen worden.

6. Test het programma door op *Act* en (eventueel) *Run* te klikken. Werkt de programma niet correct? Volg de stappen in opdracht 2 'Debuggen'

Voorbeeld:
De methode `methodeNaam( )` schrijft je als volgt:

```
/**
 *  Deze methode zorgt ervoor dat ...
 */
 public void methodeNaam () {
      // hier komt wat de methode moet doen

 }
```

De aanroep van `methodeNaam( )` vanuit de `act` is dan:

```
 public void act () {
      methodeNaam( );
 }
```

5.1.1 Aparte methodes

In het laatste onderdeel van opdracht 3 heb je code geschreven waarmee Mimi om meerdere hekjes heen kon lopen op zoek naar haar ei. Die code staat nu in `void act( )`. We maken daar nu een aparte methode van. Dit maakt de code overzichtelijker.

1. Open het scenario waar je in opdracht 3 aan gewerkt hebt, dus een kopie van `Opdr3_jouwNaam`. Bij voorkeur werk je met jouw eigen code verder. Is dat écht onmogelijk, dan mag je gebruik maken van het scenario **'DodoScenario4'**.

2. Kijk even terug naar jouw code. Wat is het doel van de code die je in `act` geschreven hebt? Bedenk welk probleem die oplost en in welke gevallen je oplossing werkt.

3. Bedenk een geschikte naam voor jouw methode. Let hierbij op de naamgevingsafspraken (zie opdracht 1 'Naamgevingsafspraken').

4. Schrijf het skelet van jouw methode (dus de signatuur, accolades en commentaar).

5. Verplaats de code uit `act` naar de body van jouw nieuwe methode.

6. Compileer en test de methode. Testen van een methode doe je door met de rechtermuisknop op het object te klikken en deze (eventueel meerdere keren achtereenvolgens) aan te roepen. Werkt die niet zoals verwacht? Volg de stappen in opdracht 2 'Debuggen'. Pas als je zeker weet dat jouw methode werkt zoals verwacht wordt, ga je door.

7. Ga terug naar de code en roep jouw nieuwe methode aan vanuit `act`.

8. Compileer en test het programma door op *Act* te drukken. Werkt die niet zoals verwacht? Volg de stappen in opdracht 2 'Debuggen'.

Je hebt nu een nieuwe methode gemaakt die je vanuit `act` aanroept. Als je meer opdrachten aan `act` wilt toevoegen, doe je dat op dezelfde wijze. Zo blijft de code overzichtelijk.

5.2 Algoritmisch denken

We gaan Mimi nu leren wat ingewikkeldere taken uit te voeren. Daarbij moet ze goed om zich heen kijken en beslissen wanneer ze iets uitvoert. Jij gaat algoritmes opstellen om haar dat te leren. In deze opdrachten voeren we het algoritme steeds uit door op de *Run*-knop te drukken. Dit houdt dus in dat de `act`-methode net zo lang wordt aangeroepen totdat de methode `Greenfoot.stop()` is uitgevoerd. In de `act`-methode zelf hoeven we daarom geen **`while`**-loop te gebruiken om de stappen van onze algoritmes te herhalen; zo'n herhaling zit immers al in de *Run*. Dat wil niet zeggen dat we geen enkele **`while`**-loop mee tegen zullen komen. Het kan immers voorkomen dat we een herhaling binnen een herhaling nodig hebben. De binnenste van die twee zullen we dan ook nu in `act` tegenkomen.

5.2.1 Eitje boven hek leggen

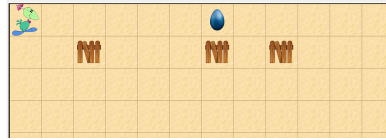
Mimi loopt door de wereld van linksboven naar rechtsboven. Als ze boven een hek staat, moet ze daar een eitje leggen. Let er op dat jouw algoritme generiek is. Het moet dus werken voor alle vergelijkbare werelden, dus ook als een hekje één plekje naar links verschoven wordt.

1. Open de wereld: 'world_layEggAboveFence'.
2. Omschrijf de begin- en eindsituaties.
3. Bedenk een algoritme.
4. Klik met de rechtermuisknop op Mimi. Controleer door het aanroepen van de methodes of de strategie gaat werken. Tip: kijk ook bij de methodes van `Dodo`.
5. Teken het bijbehorende stroomdiagram. Maak daarbij gebruik van **`if`**-constructies (in plaats van **`while`**). Kijk eventueel terug naar de uitleg en afspraken in Opdracht 3 over het vermijden van een **`while`** in de `act`.
6. Bedenk een geschikte methodenaam. Let hierbij op de naamgevingsafspraken.
7. Schrijf de bijbehorende methode.
8. Schrijf daar ook commentaar bij.
9. Compileer en test jouw methode door met de rechtermuisknop op Mimi te klikken en de methode te selecteren. Werkt die niet zoals verwacht? Volg de stappen zoals beschreven in hoofdstuk 'Debuggen' van opdracht 2.
   Tip: Wil je dat steeds een bepaalde wereld geopend wordt, volg dan de volgende stappen:
   (a) Klik in de klassendiagram met de rechtermuisknop op 'Madagaskar'.
   (b) Kies 'Open Editor'
   (c) Bovenin staat de volgende declaratie:
   ```
   private static String WORLD_FILE = "world_bestandsnaam.txt";
   ```
   (d) Vervang de bestandsnaam van de wereld (hetgene vóór de '.txt') met de wereld die je wilt, in dit geval 'world_layEggAboveFence'. Het ziet er dan zo uit:
   ```
   private static String WORLD_FILE = "world_layEggAboveFence.txt";
   ```
10. Roep de methode in de code aan vanuit de `act`-methode van `MyDodo`.
11. Compileer en test jouw programma.

Je hebt nu zelf code voor een genest **`if`**-statement geschreven.

5.2.2 Geen dubbele eitjes

Mimi moet natuurlijk alleen een eitje leggen als dat er nog niet ligt. We breiden de vorige opgave daarmee uit. Dus: Mimi loopt van linksboven naar rechtsboven. Als ze boven een hek staat en er ligt nog geen ei, dan legt ze daar een eitje.

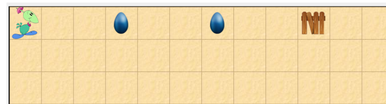


1. Open de wereld: 'world_layEggAboveFenceWithEgg'.
2. Omschrijf de begin- en eindsituaties.
3. Hoeveel eitjes moet Mimi gaan leggen? Hoe kun je nagaan hoeveel ze er gelegd heeft?
4. Bedenk een algoritme.
5. Klik met de rechtermuisknop op Mimi en controleer door het aanroepen van de methode of het algoritme gaat werken. Tip: kijk ook bij de methodes van `Dodo`.
6. Teken het bijbehorende stroomdiagram.
7. Schrijf de bijbehorende methode.
8. Schrijf daar ook commentaar bij.
9. Compileer en test jouw methode door met de rechtermuisknop op Mimi te klikken en deze te selecteren. Tip: Wil je bij het testen dat steeds een bepaalde wereld geopend wordt? Zie bij de vorige opgave (opgave 5.2.1, onderdeel 9) hoe je dat doet.
10. Roep de methode aan vanuit de `act`-methode van `MyDodo`.
11. Compileer en test jouw programma.

Je hebt nu logische operatoren en **`boolean`** methodes gebruikt om een complexere conditie op te stellen.

5.2.3 Maak een spoor van eitjes

Bekijk het volgende scenario. Mimi gaat nu een spoor eitjes achter laten totdat ze bij het hek komt. Als er al een ei ligt hoeft ze er natuurlijk niet nog een te leggen.



1. Open de wereld: 'world_spoorEitjesLeggenTotHek'.
2. Bedenk een methodenaam die het doel van Mimi goed omschrijft.
3. Er liggen nu twee eitjes. Hoeveel eitjes moet Mimi nu nog gaan leggen?
4. Omschrijf de begin- en eindsituaties.
5. Bedenk een algoritme om dit te doen.

6. Teken het bijbehorende stroomdiagram. Let er op dat er zo min mogelijk 'dubbel' gebeurt in het schema. Kijk eventueel terug bij opdracht 3, hoofdstuk **??**.
7. Schrijf de bijbehorende methode.
8. Schrijf daar ook commentaar bij.
9. Compileer en test jouw methode door met de rechtermuisknop op Mimi te klikken en deze te selecteren. Werkt de methode correct? Worden er echt op alle juiste plekken eieren gelegd? Tip: Als er in een cel twee eieren liggen, en je versleept één daarvan naar een andere cel, dan zie je het tweede ei liggen.

Je hebt nu gebruik gemaakt van nesting van statements in combinatie met verschillende condities. Mimi is hierdoor in staat om wat ingewikkeldere taken uit te voeren.

5.2.4 Loop door een tunnel

Bekijk het onderstaand scenario. Mimi gaat nu door de tunnel van hekjes lopen. Als ze de tunnel voorbij is moet ze stoppen.

We delen dit probleem eerst op in deelproblemen:

1. We maken eerst een submethode die bepaalt of Mimi naast een hek staat.
2. Daarna gebruiken we die methode om te bepalen of er een hek aan iedere kant van Mimi staat (en dus of Mimi in een tunnel staat). Dan weten we namelijk of ze nog een stap moet zetten of niet.

Nu pakken we één voor één ieder deelprobleem aan:

1. **Staat Mimi naast een hek?**: We maken eerst een submethode om te bepalen of Mimi naast een hek staat. Je schrijft hiervoor een eigen **`boolean`** methode. Deze moet teruggeven of er links van Mimi een hek staat of niet. In opdracht 2 'Accessormethode' staat de **`boolean`** methode uitgelegd. Kijk zo nodig even terug.

   (a) Bedenk een algoritme om te bepalen of er links van Mimi een hek staat.

   (b) Teken het bijbehorende stroomdiagram.

   (c) Bepaal de begin- en eindsituaties. Zijn deze hetzelfde?

   (d) Schrijf de bijbehorende methode **`boolean`** `fenceOnLeft()` die **`true`** oplevert als er links van Mimi een hek staat, en anders **`false`**.

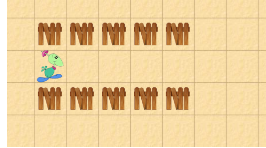
   (e) Schrijf daar ook commentaar bij.

   (f) Compileer en test het jouw methode door met de rechtermuisknop op Mimi te klikken en de methode aan te roepen. Geeft de methode in de juiste gevallen **`true`** en **`false`** terug?

   (g) Zijn de begin- en eindsituaties inderdaad gelijk? Dat wil in ieder geval zeggen: Mimi staat op dezelfde plek, met haar snavel in dezelfde richting. Er is verder ook niks aan het scenario veranderd.

   (h) Schrijf op een vergelijkbare manier een methode **`boolean`** `fenceOnRight()`. Omdat deze erg veel op **`boolean`** `fenceOnLeft()` lijkt hoef je geen nieuw stroomdiagram te tekenen. Wel moet je de methode nog zorgvuldig testen.

2. **Loop tot einde van de tunnel**: Mimi gaat nu door de tunnel van hekjes lopen. Als ze de tunnel voorbij is moet ze stoppen.



   (a) Open de wereld: 'world_walkThroughTunnel'.
   (b) Bedenk een algoritme om dit te doen. Je mag natuurlijk gebruik maken van jouw methoden **boolean** `fenceOnLeft()` en **boolean** `fenceOnRight()`.
   (c) Teken het bijbehorende stroomdiagram. Probeer hierbij zo min mogelijk blokjes en ruitjes te gebruiken. Tip: Maak bij de 'keuze' gebruik van logische operatoren, zie 4.
   (d) Bepaal de begin- en eindsituaties.
   (e) Schrijf een methode **void** `walkThroughTunnel()` waarmee Mimi door de tunnel van hekjes loopt.
   (f) Schrijf daar ook commentaar bij.
   (g) Compileer en test jouw methode.
   (h) Roep deze methode aan vanuit `act`.
   (i) Compileer en test het programma.

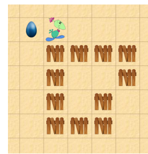
Je hebt nu zelf gebruik gemaakt van abstractie om een grotere probleem op te splitsen in kleinere delen. Met het verdeel-en-heers principe heb je de kleinere delen ontworpen, geïmplementeerd en getest (de `fenceOnRight()` en `fenceOnLeft()`). Daarna heb je deze gebruikt in `walkThroughTunnel()` en het geheel getest. Wie weet, misschien wil je straks in een volgende opgave gebruik maken van een van jouw methodes (bijvoorbeeld `fenceOnRight()`). Dat kan je nu makkelijk doen. Het is een aparte methode die je al goed getest hebt. Dat heet *modularisatie*.

5.3 Om een omheining lopen

We leren Mimi om een omheining te lopen. Ook hier maken we een methode die Mimi steeds hooguit één stap laat zetten. Door op *Run* te drukken voeren we het algoritme stapsgewijs uit. Om te voorkomen dat Mimi oneindig veel rondjes om het hek heen wandelt gaan we ervan uit dat in het laatste vakje van de route om de omheining een ei ligt. Als ze gaat lopen en haar eitje tegenkomt, dan weet ze dat ze klaar is.

1. Open de wereld: 'world_walkAroundFencedArea'.
2. Bedenk een algoritme om dit te doen.
3. Teken het bijbehorende stroomdiagram. Tip: Maak daarbij gebruik van methodes die je al geschreven en getest hebt in de vorige opgave.
4. Bepaal de begin- en eindsituaties.
5. Schrijf de bijbehorende methode **void** `walkAroundFencedArea()` waarmee Mimi om de omheining heen loopt.

6. Schrijf daar ook commentaar bij.
7. Compileer en test jouw methode.
8. Roep vanuit `act` de methode aan.
9. Compileer en test het programma door op de *Run*-knop te drukken.
10. Controleer ook of jouw programma werkt voor een iets grotere omheining.
11. Test in welke beginsituaties jouw programma werkt. Zet Mimi op een andere positie neer. Werkt jouw programma dan nog steeds goed? Pas de 'beginsituatie'-tekst in het stroomdiagram eventueel aan.
12. Test ook of jouw methode werkt met een andere wereld, zoals: 'world_walkAroundOtherFencedArea'. Pas het algoritme (en dus ook het stroomdiagram en code) dusdanig aan dat het ook hiervoor werkt.



Figuur 1: Een andere omheining

Je hebt nu een generieke methode geschreven waarmee Mimi om een willekeurige omheining kan lopen. Je hebt daarbij gebruik gemaakt van modularisatie door (bestaande en geteste) methodes te hergebruiken.

Uitlijning code
Code wordt beter leesbaar door steeds dezelfde stijl te gebruiken. Net als naamgeving, zijn er ook stijlafspraken. Het gebruik van inspringen voorkomt dat je fouten maakt. Zo zie je sneller of je een haakje of accolade te veel of te weinig hebt. Greenfoot kan jouw code voor je uitlijnen. Gebruik hiervoor Ctrl-Shift-I.

Zie `http://google-styleguide.googlecode.com/svn/trunk/javaguide.html` voor een compleet overzicht van stijl- en naamgevingsafspraken.

5.4 Eitjes volgen tot het nest

Mimi gaat een spoor van eitjes volgen totdat ze bij het einde komt. We delen het probleem op in deelproblemen:

- Bepaal of het nest is gevonden;
- Volg het pad van eitjes.

We lossen het probleem nu stapsgewijs op door de deelproblemen los van elkaar op te lossen.

1. Open de wereld: 'world_followEggTrailUntilNest'.

2. **Bepaal of het nest is gevonden**: Bedenk een strategie voor Mimi om te bepalen of ze het nest gevonden heeft.

   (a) Bedenk een algoritme voor het bepalen of in het vakje rechts of links van Mimi een nest ligt. Tip: maak gebruik van de methode **boolean** `nestAhead( )` van `Dodo`.

   (b) Wat voor een soort methode gaat dit worden (**void** / **boolean** / **int**)?

   (c) Teken hiervoor het bijbehorende stroomdiagram. Let erop dat er na het aanroepen van een **return** geen code meer kan worden uitgevoerd. In een stroomdiagram wordt een 'return' altijd direct opgevolgd door een 'Einde'. Als Mimi nog iets moet doen, dan moet dat daar voorafgaand aan gebeuren.

   (d) Bepaal de begin- en eindsituaties.

   (e) Kloppen de begin- en eindsituaties met de soort methode die je gebruikt hebt? (Tip: een **boolean** methode moet eigenlijk niks aan de situatie veranderen, Mimi moet dus op dezelfde plek staan met haar snavel in de zelfde richting).

   (f) Schrijf de bijbehorende methode.

   (g) Schrijf daar ook commentaar bij.

   (h) Compileer en test jouw methode door deze met de rechtermuisknop aan te roepen. Probeer verschillende scenario's uit.

3. **Volg pad van eitjes tot nest**: Bedenk een algoritme waarmee Mimi het pad van eitjes tot het nest volgt. Tip: als je merkt dat dit al gauw ingewikkeld wordt, volg dan de volgende stappen:

   (a) Om te bepalen of er een eitje voor Mimi ligt kan ze **boolean** `eggAhead( )` gebruiken. Test de `Dodo` methode om te bekijken wat die precies doet.

   (b) Bedenk een algoritme om te bepalen of er in het vakje links van Mimi een ei ligt. Tip: Maak daarbij gebruik van de methode **boolean** `eggAhead( )` van `Dodo`.

   (c) Wat voor een soort methode gaat dit worden (**void** / **boolean** / **int**)?

   (d) Teken hiervoor het bijbehorende stroomdiagram. Houd er wederom rekening mee dat er na een 'return' niets meer uitgevoerd kan worden.

   (e) Bepaal de begin- en eindsituatie.

   (f) Schrijf de bijbehorende methode.

   (g) Kloppen de begin- en eindsituaties met de soort methode dat je gebruikt hebt?

(h) Compileer. Test je code door met de rechtermuisknop op Mimi te klikken. Probeer verschillende situaties uit.

(i) Schrijf en test een vergelijkbare methode om te bepalen of er in het vakje rechts van Mimi een ei ligt.

4. Teken een stroomdiagram. Verwijs daarbij naar de twee methodes die je zojuist gemaakt hebt.
5. Bepaal de begin- en eindsituatie.
6. Schrijf een methode `void followEggTrailUntilNest()` waarmee Mimi het spoor van eieren volgt.
7. Schrijf daar ook commentaar bij.
8. Compileer en test de methode `void followEggTrailUntilNest()`.
9. Roep vanuit de `void act( )` de methode aan.
10. Compileer en test het programma door op de *Act*-knop te drukken.
11. Controleer ook of jouw programma ook werkt voor een ander pad van eieren. Test deze ook met 'world_followEggTrailBehindUntilNest'. Wellicht kom je tot de ontdekking dat je nog wat moet uitbreiden.

5.5 Kwaliteit

Wat is een goede oplossing?
Of een oplossing 'goed' is bepaal je o.a. aan de hand van de volgende *kwaliteitscriteria voor programma's*:

- Correctheid: Het algoritme doet wat het moet doen. Het doet niet wat het niet moet doen.
- Efficiëntie: snelheid en gebruik van middelen (processor, geheugen, netwerk, etc.) zijn geschikt voor het oplossen van het probleem.
- Elegantie/slimheid: Het is generiek (voor meerdere problemen toepasbaar).
- Schaalbaarheid/aanpasbaarheid: Het kan eenvoudig uitgebreid en aangepast worden. Er is gebruik gemaakt van modularisatie en abstractie.
- Betrouwbaarheid: De programma loopt niet vast, ook niet bij onverwachte input.
- Onderhoudbaarheid: Er is gebruik gemaakt (naamgevings)afspraken, commentaar, logische begin- en eindsituaties, modularisatie en abstractie.
- Bruikbaarheid: Het is gebruikersvriendelijk (o.a. er zijn geschikte foutmeldingen).

Wat is goede code?
De programmacode dient te voldoen aan de volgende *kwaliteitscriteria voor code*:

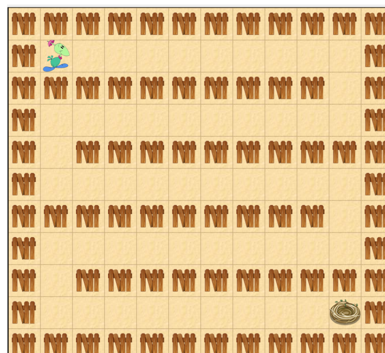
- Leesbaar: uit de code moet duidelijk zijn wat het beoogde doel van de code is, de code voldoet aan de (naamsgevings)conventies, en is opgebouwd uit modules.
- Testbaar: de code moet dusdanig zijn opgebouwd dat deze in delen te testen is.
- Flexibel: zo min mogelijk afhankelijkheden
- Correct: het doet wat er van verwacht wordt.
- Efficiënt: het gaat zuinig om met geheugen en tijd.

We kijken nu even terug naar het programma dat je in de laatste opgave geschreven hebt. Beoordeel jouw eigen oplossing aan de hand van een aantal kwaliteitscriteria. Voor de volgende punten ge je na in hoeverre jouw oplossing hieraan voldoet en leg je uit waarom je dat vindt.

1. Aan welke van de volgende kwaliteitscriteria voor **programma's** voldoet jouw oplossing? Leg uit.
   - Correctheid:
   - Elegantie/slimheid:
   - Schaalbaarheid/aanpasbaarheid:
   - Betrouwbaarheid:
   - Onderhoudbaarheid:
2. Aan welke van de volgende kwaliteitscriteria voor **code** voldoet jouw oplossing? Leg uit.
   - Leesbaar:
   - Testbaar:
   - Correct:
3. Nu heb je zelf al wat code geschreven. Welke van de kwaliteitscriteria vind jij het belangrijkst? Waarom?
4. Bekijk de code van de laatste opgave van een medestudent. Wat valt jou op? Bespreek dat samen.
5. Wat voor soort fouten heb je in deze opdracht gemaakt? Waarom was dat? Wat heb je gedaan om die op te lossen?

5.6 Doolhof

Help jij Dodo haar nest te vinden? Hiervoor moet je haar helpen om haar weg door de doolhof te vinden. Jouw oplossing moet natuurlijk generiek zijn. Dodo moet haar weg door een willekeurige doolhof vinden.



5.6.1 Simpele doolhof (A)

Je mag van het volgende uitgaan:

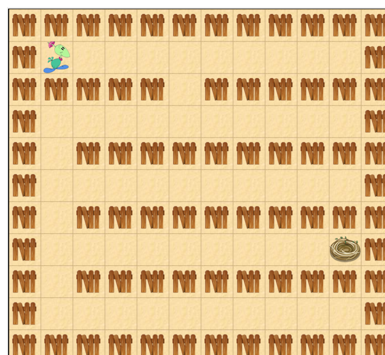
- De wereld is omgeven door een hek (fence).
- Er is één nest.
- Er bestaat een route naar het nest.
- Het nest ligt direct naast een hek.
- Er is hooguit één route mogelijk (er zijn geen doodlopende gangen en ook geen 'eilanden' van hekjes).
- Dodo kan vanuit de beginsituatie maar één kant op.

Schrijf een methode waarmee Dodo het nest vindt.

1. Bedenk een algoritme.
2. Bedenk welke submethodes handig zijn. Beschrijf, implementeer en test deze één voor één.
3. Teken een stroomdiagram, schrijf de bijbehorende code en test jouw programma op dezelfde manier als dat je bij de vorige opgaven gedaan hebt.
4. Geef Dodo ook een complimentje als ze haar nest gevonden heeft.
5. Beoordeel jouw programma. Welke verbeteringen kan je voorstellen?
6. Test jouw programma in de volgende doolhoven:
    - 'world_doolhofLevel1a'.
    - 'world_doolhofLevel1b'.
    - 'world_doolhofLevel1c'.

5.6.2 Lastigere doolhof

We maken de doolhof moeilijker.



Je mag van het volgende uitgaan:

- Er is één nest.
- Er bestaat een route naar het nest.
- De wereld is omgeven door een fence.

- Het nest ligt direct naast een fence.
- Er zijn geen 'eilanden' van hekjes.

Schrijf een methode waarmee Dodo het nest vindt.

1. Schrijf een methode op dezelfde manier als je gedaan hebt bij de vorige opgaven.
2. Test jouw programma in de volgende doolhoven:
   - 'world_doolhofLevel2a'.
   - 'world_doolhofLevel2b'.
   - 'world_doolhofLevel2c'.
3. Maak zelf ook een nieuwe doolhof. Test jouw programma daarmee.
4. Beoordeel jouw programma. Welke verbeteringen kan je voorstellen? Beoordeel jouw code en programma ook aan de hand van de kwaliteitscriteria.
5. Wissel jouw doolhof uit met een medeleerling. Test elkaars programma's op elkaars doolhoven. Vergelijk elkaars oplossingen.

Hint: In één van de vorige opgaven heb je een 'trucje' gebruikt om te beoordelen of Mimi al eens eerder op die plek geweest was.

6 Samenvatting

Je hebt geleerd:

- een complex probleem op te delen in deelproblemen en deze afzonderlijk aan te pakken (abstractie);
- deelproblemen te ontwerpen, implementeren en te testen (modularisatie);
- submethodes vanuit andere methodes aanroepen (hergebruiken);
- **`boolean`** methodes op te stellen en gebruik te maken van return statements;
- `!`, &&, || en **`boolean`** methodes te gebruiken in condities;
- gebruik te maken van nesting;
- stroomdiagrammen en code te beoordelen op kwaliteit.

7 Jouw werk opslaan

Je bent klaar met de vierde 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 opgavenummer, bijvoorbeeld: `Opdr4_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 ...'.