Assignment 4: Generic solution

– Algorithmic Thinking and Structured Programming (in Greenfoot) –

©2015 Renske Smetsers-Weeda & Sjaak Smetsers
Licensed under the Creative Commons Attribution 4.0 license,
https://creativecommons.org/licenses/by/4.0/

1 Introduction

In the previous assignments you came up with your own solutions and implemented those. You als learned how to make generic solutions. In this assignment you will teach Mimi to do more complex tasks. In order to do this, you will have to integrate the things that you have learned in the previous assignments.

2 Learning objectives

After completing this assignment, you will be able to:

- use **logical operators**: `!`, && and ||;
- write your own **`boolean`** method;
- use combinations of logical operators and **`boolean`** methods to develop complex **conditional expressions**;
- apply **nested `if`** .. `then` .. **`else`** statements;
- use **`return`** statements in flowcharts and in code;
- use a flowchart to develop a complex algorithm;
- apply method calls and make use of result-types;
- make proper use of the naming conventions for methods;
- apply **modularization**: design, write, test, and call sub-methods individually;
- determine if an accessor method has equal initial and final situations;
- name **quality criteria** for programs and code;
- describe the advantages of designing a program in a structured manner on (the types and amount of) implementation errors;
- determine if a program and its code meet certain quality requirements;
- find and analyse errors in code, interpret error messages and use this to fix problems (debugging).

3 Instructions

In this assignment you will carry on with your code from assignment 3. Therefore you will need a copy of the scenario which you saved after completing assignment 3, `Asgmt3b_yourName`. To make a copy follow the next steps:

- Open your scenario from assignment 3, `Asgmt3b_yourName`.
- In the Greenfoot menu at the top of the screen, select 'Scenario' and then 'Save As ...'.
- Check that the window opens in the folder where you want to save your work.
- Choose a file name containing your own name(s) and assignment number, for example: `Asgmt4_John`.

You will also need answer some questions. Questions with an '(IN) ' must be handed 'IN'. For those you will need:

- pen and paper to draw flowcharts which must be handed in ('(IN) '),
- a document (for example, Word) open to type in the answers to the '(IN) ' questions.

The answers to the other questions (those that don't need to be handed in) you must discuss with your programming partner and then jot down a short answer on the assignment paper.

Note: We recommend that you to continue working with your own code. If it is **absolutely** impossible to carry on working with your own code from assignment 3, then you may download and use **'DodoScenario4'**.

4 Theory

Logical operators
Java has three logical operators, shown in the table below. You have already seen (and used) a few of them. You can combine these with **`boolean`** methods to create a *conditional expression* which describes a particular selection (i.e. choice).

Operator	Meaning	Example
&&	AND	`facingNorth() && fenceAhead()`
||	OR	`fenceAhead() || borderAhead()`
!	NOT	`!fenceAhead()`

5 Exercises

In the following exercises we will teach (i.e. program) Mimi to do all kinds of things. It is important to make Mimi be as smart as possible. We will teach her to do new things, but want her to be able to do those in any similar situation she is faced with in the future. So, we're looking for *generic solutions* for the problems she runs into.

For example, if Mimi has to find her egg in the scenario above, then we don't want to teach her to "take 3 steps". That won't help her if in a future situation where the egg may be one step further away. We would rather say: "While not found egg, take a step." This will work in any similar situation, whether the egg is 2 or 38 steps away.

5.1 Own functionality: separate method

Modularization
Until now you have written all your code in the `act` method. However, if you want to expand your code any further, then at a certain moment it will become cluttered, hard to read and therefore hard to maintain or modify. Another disadvantage is that in each exercise you keep replacing your code with new code. You don't have your 'old' code any more. That's a shame, it may be useful to re-use things you have already done.

It would be better to write code for each subtask (or exercise) in a separate method. This can be tested individually, and easily re-used. This is called modularization.

Steps for applying modularization:

1. Determine the algorithm.
2. Draw the corresponding flowchart.
3. Write the corresponding code. Stick to the naming conventions (see assignment 1 'Naming conventions').
4. Test the method individually. Do this by right-clicking on the object in the world and selecting the method. Does the method not work as expected? Follow the steps for 'Debugging' in assignment 2. Only when you are absolutely convinced that your method works properly, you can continue.
5. Modify the `act( )` so that this calls your new method. You can also call your method in another method.
6. Test the program by pressing *Act* (and possibly *Run*). Does the program not work as expected? Follow the steps in 'Debugging' in assignment 2.

Using modularization in code:
The new method `methodName()` is written as follows:

```
/**
 *  This method ensures that ...
 */
 public void methodName () {
      // here you write the code for the method

 }
```

With the following code you call the `methodName ( )` method in the `act`:

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

5.1.1 Separate methods

In the last exercise of assignment 3 you wrote code so that Mimi could walk around several consecutive fences. That code was written in the **`void`** `act( )` method. We are now going to move that code to a separate (new) method. This will make the code clearer.

1. Have a look at your code in `MyDodo`'s `act` method. What is the purpose of the code? Which problem does it solve? Consider in which cases the solution works.

2. Think of a suitable name for your method. In doing so, keep to the naming conventions (see assignment 1 'Naming conventions').

3. Write the skeleton of your method (thus the signature, curly brackets and comment).

4. Move the code from the `act` method to the body of your new method (so, in between the curly brackets).

5. Compile and test the method. You can separately test a method by right-clicking on the object and then selecting the method (if needed, several times consecutively). Does the program not work as expected? Follow the steps in 'Debugging' in assignment 2. Only when you are absolutely convinced that the method works accordingly, you may carry on.

6. Go back to the code and call the new method in the `act` method.

7. Compile and test the program by pressing *Act*. Does the program not work as expected? Follow the steps in 'Debugging' in assignment 2.

You have now made a new separate method which you called in the `act`. If you want Mimi to do more tasks, you can add more method calls in the `act` in the same way. This keeps the code clear and easy to follow.

5.2 Algorithmic thinking

We are now going to teach Mimi to do more complicated tasks. In order to do that, she must properly look around and decide what she should do in which situations. It's your task to come up with suitable algorithms for teaching her that. In the following exercises we will execute the algorithm by pressing *Run*. This therefore means that the `act` method will continuously be repeated until `Greenfoot.stop( );` is called. Because of this, we don't need a **`while`** loop to repeat the steps of our algorithm; such a repetition has been built into Greenfoot's *Run*. However, that doesn't mean we won't be using any **`while`** loops anymore! We may need to construct a repetition within a repetition (in this case the second repetition may be a **`while`** within the `act`).

5.2.1 Lay an egg above each fence

Your mission is as follows: "Mimi walks across the world, from the top-left corner to the top-right corner. If she finds herself above a fence, she must lay an egg."
Make sure your algorithm is generic. It must work in any similar world, so also if one of the fences is moved one position to the right or left.

1. Open the 'world_layEggAboveFence' world.

2. Describe the initial and final situations.

3. Come up with a suitable algorithm.

4. Right-click on Mimi. Check, by calling the methods in your algorithm, if your strategy will work. Tip: also have a look at the inherited `Dodo`'s methods.

5. Draw the corresponding flowchart. Make use of the **`if`** .. `then` .. **`else`** construct (instead of **`while`**). If you wish, have a look back at the explanation and agreements we made in assignment 3 about avoiding a **`while`** in the `act` method.

6. Think of a suitable name for your method. In doing so, keep the naming conventions in mind (see assignment 1 'Naming conventions').
7. Write the corresponding code (and comments) for your method.
8. Compile and test your method by right-clicking on Mimi in the world, and then selecting the method. Does the program work as expected?
   Tip: if you want to have Greenfoot open a particular world each time (in this case 'world_layEggAboveFence'), then follow the next steps:
   (a) Right-click on 'Madagaskar' in the class diagram.
   (b) Select 'Open Editor'
   (c) Near the top you will find the following declaration:

   ```
   private static String WORLD_FILE = "world_bestandsnaam.txt";
   ```

   (d) Replace the filename (the part in front of '.txt') with the world you wish to open, in this case 'world_layEggAboveFence'. It should then look like this:

   ```
   private static String WORLD_FILE = "world_layEggAboveFence.txt";
   ```

9. Call your new method in `MyDodo`'s `act` method.
10. Compile and test your program.

You have now written your own nested **if** statement.

5.2.2 (IN) No double eggs

Obviously, Mimi should only lay an egg if there is no egg there yet. We will modify the code from the previous exercise.
The new mission is: "Mimi walks across the world, from the top-left corner to the top-right corner. If she finds herself above a fence and there is no egg there yet, then she must lay an egg."

1. Open the 'world_layEggAboveFenceWithEgg' world.
2. Describe the initial and final situations.
3. How many eggs should Mimi lay? How can you determine how many eggs she has layed?
4. Devise a suitable algorithm.
5. Right-click on Mimi. Check, by calling the methods in your algorithm, if your strategy will work. Tip: also have a look at `Dodo`'s methods.
6. Draw the corresponding flowchart.
7. Write the corresponding code (and comments) for your method.
8. Compile and test your method by right-clicking on Mimi in the world, and then selecting the method. Does the program work as expected? Tip: Do you want to continuously test in a particular world, follow the steps in the previous exercise (exercise 5.2.1, part 8).
9. Call your new method in `MyDodo`'s `act` method.
10. Compile and test your program.

You have now used logical operators and **boolean** methods to make a complex conditional expression.

5.2.3 Laying a trail of eggs

Have a look at the scenario below. Mimi must now lay a trail of eggs, and stops when she reaches a fence. Obviously, if there is already an egg, she doesn't need to lay another one.

1. Open the 'world_layEggTrailUntilFence' world.
2. Determine a method name which adequately describes Mimi's goal.
3. There are two eggs in the scenario. How many eggs does Mimi still have to lay?
4. Describe the initial and final situations.
5. Come up with an algorithm to do this.
6. Draw the corresponding flowchart. While doing so, keep in mind what you learned about redundancy (Assignment 3, exercise 5.1.1).
7. Write the corresponding method. Don't forget the comments.
8. Compile and test your method by right-clicking on Mimi. Does the method work as expected? Did Dodo lay eggs in the right places? Are you sure she didn't lay an egg where there was already an egg? Tip: To check if there are two eggs in a square, drag the egg to another position. If you see that a second egg is revealed in that square, then Mimi is doing something wrong.

Mimi is now capable of carrying out more complex tasks. You combined the nesting of statements with conditional expressions to achieve this.

5.2.4 (IN) Walk through a tunnel

Have a look at the scenario below. Your mission is to teach Mimi to walk through a tunnel of fences, and stop when she gets clear of the tunnel (i.e. has passed the tunnel).

First we break the problem down into two sub-problems:

1. We first make a sub-method which checks whether Mimi has a fence to her left or right.
2. We then use that sub-method to check whether Mimi has a fence on both (left and right) sides. In that case, she is standing in a tunnel (and we know that she has to take at least one more step).

Step-by-step, we solve each of the two sub-problems:

1. **Is Mimi standing next to a fence?**: First we develop a sub-method to determine whether Mimi is standing next to a fence. To do this you will write your own **`boolean`** method which determines if there is a fence to her left or not (afterwards, we will do the same for her right). Tip: The **`boolean`** method was explained in assignment 2, section 'Accessor method'. If needed, review that section again.

(a) Determine an algorithm which checks if Mimi has a fence on her left.

(b) Draw the corresponding flowchart.

(c) Determine the initial and final situations. Are these identical?

(d) Write the corresponding method **boolean** `fenceOnLeft()` which returns **true** when Mimi has a fence on her left, and otherwise **false**.

(e) Include comments in your code.

(f) Compile and test your method by right-clicking on Mimi. Does the method return the correct result in both cases?

(g) Are the initial and final situations indeed identical? This means: Mimi is still standing on the same position, facing the same direction. Nothing else in the scenario is changed by calling your method.

(h) Similarly, write the method **boolean** `fenceOnRight()`. Because this one is so similar to **boolean** `fenceOnLeft()` you don't have to draw the flowchart. However, you must still properly test it! You would be surprised what kind of silly mistakes you could make... When you are absolutely convinced that both methods work accordingly, you can carry on.

2. **Walk to the end of the tunnel**: Now, the second sub-problem: Mimi must walk to the end of a tunnel of fences. She's done when she is clear of the tunnel (i.e. has passed it).

(a) Open the 'world_walkThroughTunnel' world.

(b) Think up an appropriate algorithm. Make use of the methods **boolean** `fenceOnLeft()` and **boolean** `fenceOnRight()` which you have just made (and tested).

(c) Draw the corresponding flowchart. Try to use as few rectangles and diamonds as possible. Tip: Use logical operators in your conditional expressions (choices), see chapter 4 above.

(d) Determine the initial and final situations.

(e) Write a method **void** `walkThroughTunnel()` which makes Mimi walk through a tunnel of fences. Include comments in your code.

(f) Compile and test your method.

(g) Call your method in the `act`-method.

(h) Compile and test your entire program.

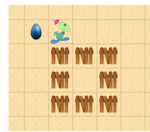
You have now applied abstraction to break a large problem down into smaller parts. Using the divide-and-conquer principle, you have designed, implemented, and tested sub-problems (`fenceOnRight()` and `fenceOnLeft()`). You used your solution to the sub-problems in `walkThroughTunnel()` and tested your program as a whole.

Who knows, maybe you can use one of your methods (such as `fenceOnRight()`) in the following exercise. That would be convenient, after all, you've tested the methods and are convinced they work properly. This is called *modularization*. It's easy and could save you time and effort.

5.3 Walk around a fenced area

Your mission is to teach Mimi to walk around a fenced area. Again, your method should make Mimi take one step only. Then, by pressing *Run* the algorithm is executed repeatedly, making her walk all the way around the entire fenced area.

To avoid Mimi walking around the fenced area for ever, we will assume that there is an egg in the very last square on her route. Then, while walking, if she finds the egg she will know she's done.



1. Open the 'world_walkAroundFencedArea' world.
2. Come up with a suitable algorithm.
3. (IN) Draw the corresponding flowchart. Tip: Use methods which you wrote and tested in the previous exercise.
4. Determine the initial and final situations.
5. Write the corresponding method **void** `walkAroundFencedArea()` which makes Mimi walk around the fenced area. Also add comments.
6. Compile and test your method.
7. Call your method in the `act` method.
8. Compile and test your program using *Run*.
9. Also check if your program works with a larger fenced area.
10. Test in which initial situations your program works. Place Mimi in another position. Does your program still work as expected? If necessary, adjust the initial situation described in your flowchart.
11. Also test if your method works in another world, such as: 'world_walkAroundOtherFencedArea'. If necessary, modify your algorithm (and thus flowchart and code) so that it also works in this world.

Figure 1: Another fenced area

You have now written a generic method with which Mimi can walk around any arbitrary fenced area. To do this, you applied modularization by re-using (existing and tested) methods.

Code alignment
Code becomes easier to read if the same style is used consistently. Just like naming conventions, there are also style conventions. See `http://google-styleguide.googlecode.com/svn/trunk/javaguide.html` for a complete list of style and naming conventions. Greenfoot can help you align your code by using Ctrl-Shift-I.

5.4 Find the nest by following a trail of eggs

Your new mission: Mimi has to follow a trail of eggs, until she reaches her nest. Once again, we break the problem down into more manageable sub-problems:

- Determine if the nest has been found;
- Follow the trail of eggs.

We solve the problem step-by-step by solving the sub-problems individually:

1. Open the 'world_followEggTrailUntilNest' world.
2. **Determine if the nest has been found**: Find a strategy for determining if Mimi has found her nest:
   (a) Come up with an algorithm for determining if there is a nest to Mimi's right or left. Tip: use `Dodo`'s method **`boolean`** `nestAhead( )`.
   (b) What kind of method (**`void`** / **`boolean`** / **`int`**) will this be?
   (c) Draw the corresponding flowchart. Recall that no code can be executed after a **`return`** statement. In in the flowchart the 'End' always immediately follows a 'return'. So, if there is something that you still need Mimi to do, it must happen before the 'return'.
   (d) Determine the initial and final situations.
   (e) Do the initial and final situations correspond to the type of method you are using? (Tip: a **`boolean`** method should not change the situation: Mimi should still be standing in the initial position and facing the initial direction).
   (f) Write the corresponding method. Also add comments to your code.
   (g) Compile and test your method by right-clicking on Mimi. Try several different situations.
3. **Follow the trail of eggs to the nest**: Make Mimi follow a trail of eggs until she reaches her nest. Tip: try to do this on your own first. If you find that things become complicated too quickly, follow the next steps:
   (a) To determine if there is an egg in front of Mimi, she can use **`boolean`** `eggAhead( )`. Try out the `Dodo` method in different situations to figure out what it precisely does.
   (b) Come up with an algorithm to figure out if there is an egg to Mimi's left. Tip: Use `Dodo`'s method **`boolean`** `eggAhead( )`.
   (c) What kind of method (**`void`** / **`boolean`** / **`int`**) will this be?
   (d) (IN) Draw the corresponding flowchart. Again, recall that no steps can be executed after a 'return'.

(e) Determine the initial and final situations.

(f) Do the initial and final situations correspond to the type of method you are using? (Tip: a **boolean** method should not change the situation, Mimi should still be standing in the same position facing the same direction).

(g) Write the corresponding method. Also add comments to your code.

(h) Compile and test your method by right-clicking on Mimi. Try several different situations.

(i) Write and test a similar method to determine if Mimi has an egg to her right.

4. (IN) Draw a flowchart. Refer to the two methods you have just written by calling them in a rectangle.
5. Determine the initial and final situations.
6. Write the method **void** `followEggTrailUntilNest()`. Add comments.
7. Compile and test your **void** `followEggTrailUntilNest()` method.
8. Call this method in the **void** `act( )` method.
9. Compile and test the program using the *Act* button.
10. Test if your program works for another trail of eggs. Also test it using the 'world_followEggTrailBehindUntilNest' world. You may discover that you still need to make some changes.

5.5 (IN) Quality

What is a good solution?
The following *quality criteria for programs* should be met:

- Correctness: it does what it should do, and it doesn't do what it shouldn't do.
- Efficient: time and resources (processor, memory, network, etc.) required are suitable for the problem.
- Elegance / cleverness: it is generic (applicable to multiple problems)
- Scalability / adaptability: it can easily be extended or modified. Abstraction and modularization have been applied.
- Reliability: the program does not crash, even with unexpected (user)input.
- Maintainability: comments, logical initial and final situations, modularisation, and abstraction are used and naming conventions are followed.
- Usability: it is user-friendly (for example, it shows appropriate error messages).

What is good code?
The code should meet the following *quality criteria for code*:

- Correctness: it does what is expected.
- Efficient: it uses memory and time sparingly.
- Readability: it is clear what the intended purpose of the code is, it complies to (style and naming) conventions, has comments, and is made up of modules.

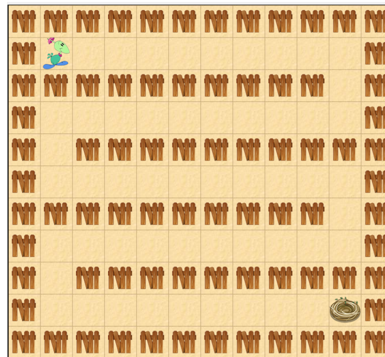
- Testability: the code is constructed such that parts can be tested individually.
- Flexibility: it has minimal dependencies (such as hard-coded values or reliant on other modules).

Let's review your program and rate you solution according to a few of the quality criteria mentioned above. For each of the following, assess how your solution meets the requirements. In each case justify your answer:

1. Your solution meets which of the following quality criteria for **programs**? Explain.
   - Correctness:
   - Elegance / cleverness:
   - Scalability / adaptability:
   - Reliability:
   - Maintainability:
2. Your solution meets which of the following quality criteria for **code**? Explain.
   - Correctness:
   - Readability:
   - Testability:
3. Now that you have written some of your own code. In your opinion, which of the quality criteria is the most important? Why?
4. Have a look at a classmate's code for the previous exercise. What do you notice? What stands out? Discuss this together.
5. Which kinds of errors did you make in this assignment? Why was that? What did you do to solve them? What can you do in the future to prevent them?

5.6 Maze

Can you help Mimi find her nest? To do this, you have to teach her to find her way through a maze. Of course, your solution has to be generic: Mimi has to be able to find her way through any arbitrary maze.



5.6.1 Simple maze

You may assume the following:

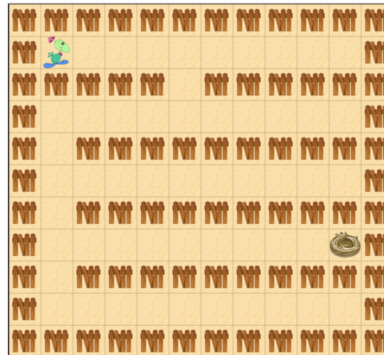
- The world is surrounded by a fence.
- There is exactly one nest.
- The nest is located next to a fence.
- A route to the nest exists.
- At most one route exists (there are no dead-ends, and no 'islands' of fences).
- From the initial position, Mimi can only walk in one direction.

Write a method helping Mimi find her nest:

1. Come up with a suitable algorithm.
2. Determine which sub-methods may come in handy. Design, implement, and test these individually.
3. Draw a flowchart, write the corresponding code and test your program, just like you did in the previous exercises. Ask yourself the same questions too!
4. Give Mimi a compliment when she finds her nest.
5. (IN) Assess your program. Which improvements can you suggest? You don't necessarily need to implement them, just list them.
6. Test your program using the following mazes:
   - 'world_doolhofLevel1a'.
   - 'world_doolhofLevel1b'.
   - 'world_doolhofLevel1c'.

5.6.2 Complex maze

We will now make the maze more complex.



You may now assume the following:

- The world is surrounded by a fence.
- There is exactly one nest.
- The nest is located next to a fence.
- A route to the nest exists.
- There are no 'islands' of fences.

Write a method helping Mimi find her nest:

1. Write a method, in the same manner as you did in the last exercises.
2. Test your program with these mazes:
   - 'world_doolhofLevel2a'.
   - 'world_doolhofLevel2b'.
   - 'world_doolhofLevel2c'.
3. Make your own maze. Test your program with it.
4. Assess your program. Which improvements can you suggest? Also use the quality criteria to rate your code and program.
5. Exchange your maze with that of a classmate. Test each other's programs on each other's mazes. Compare your solutions and give each other feedback.

Hint: In one of the previous exercises you used a 'trick' to determine if Mimi had been in that spot before.

6 Summary

You can now:

- break a complex problem down into sub-problems (abstraction);
- design, implement, and test sub-problems individually (modularization);
- call sub-methods from other methods (re-use);
- create **`boolean`** methods and make use of *return* statements;
- use `!`, &&, || and **`boolean`** methods in conditional expressions;
- make use of nesting;
- assess the quality of code and programs.

7 Saving your work

8 Saving your work

You have just finished the forth assignment. Save your work! You will need this for future assignments. In the Greenfoot menu at the top of the screen, select 'Scenario' and then 'Save'. You now have all the scenario components in one folder. The folder has the name you chose when you selected 'Save As ...'.

9 Handing in

Hand the flowcharts on paper (those indicated with an '(IN) ') into the pigeon hole outside the teachers' lounge. (You may also scan/photograph them and paste them into your (Word) document.)

Hand your (digital) work in on Magister:

1. Go to the folder where you saved your work (for example: `Asgmt4_yourName`).
2. With each scenario a 'README.TXT' file is automatically generated. Open the 'README.TXT' file and type your name(s) at the top.
3. Place the (Word) document with your answers to be handed in (answers to the '(IN) ' questions) in the same folder. Make sure your name(s) are in the document.
4. Compress the entire file into one `.zip` file. In Windows you can do this by right-clicking on the folder and choosing 'Send to' and then 'Compressed (zipped) folder'.
5. Hand the compressed (zipped) folder in via Magister.

Make sure you hand your work in before next Wednesday 8:00 (in the morning).