Assignment 3: Optimizing solutions

– 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 have become acquainted with Greenfoot and algorithmic thinking. You are able to read, modify and write your own code. You also practiced devising generic solutions and implementing them. In this exercise you will learn how to write code in a smarter way. With this cleverness, your solution will be more efficient and more elegant. It also makes it easier to re-use your solution in other situations (in other methods or programs).

2 Learning objectives

After completing this assignment, you will be able to:

- recognize nesting in a flowchart and in code;
- apply **nesting** as a strategy for solving problems;
- **optimize** a flowchart;
- modify code according to proposed changes in a flowchart;
- name benefits of **abstraction**;
- identify candidates for sub-methods in a flowchart;
- apply abstraction in flowcharts and corresponding code by using **sub-methods**;
- describe the advantages of designing a program in a structured manner, for example by using a flowchart, on (the types and amount of) implementation errors;
- determine whether a solution is **generic**;
- explain in your own words how the Greenfoot *Run* works (as a repeating *Act*);
- stop a Greenfoot program;
- write, compile, run and test your own code.

3 Instructions

In this assignment you will carry on with your code from assignment 2. Therefore you will need a copy of the scenario which you saved after completing assignment 2, `Asgmt2_yourName`. To make a copy follow the next steps:

- Open your scenario from assignment 2, `Asgmt2_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: `Asgmt3_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 2, then you may download and use **'DodoScenario3'**.

4 Theory

Nesting
The language construct for sequence, selection (choice), and repetition can also be used together in various combinations. They can be used sequentially, one after the other, or *nested* in each other.

Flowchart:
The following is an example of a flowchart of an **`if`** .. `then` .. **`else`** statement (selection) nested in a **`while`** loop (repetition).

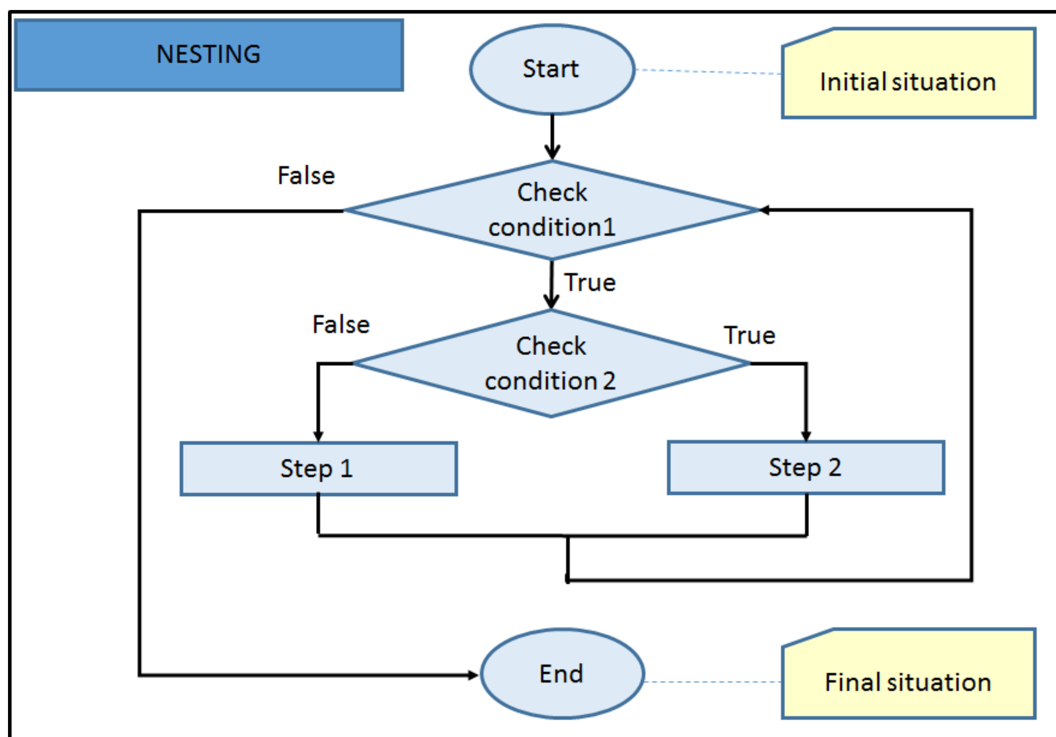


Figure 1: Flowchart for a selection nested in a repetition

The flowchart explained:

- First the conditional expression 'Check condition1' in the first diamond is evaluated.
- If the conditional expression is not true, then the 'False' arrow on the left will be followed and the method will end.
- If the conditional expression is true, then the 'True' arrow will be followed to the second diamond. The conditional expression 'Check condition2' will be evaluated.
  - If the conditional expression 'Check condition2' is true, then 'Step2' will be executed.
  - If the conditional expression 'Check condition2' is false, then 'Step1' will be executed.
- After executing 'Step1' or 'Step2', in either case we return to the first diamond (evaluate 'Check condition1'). If the expression is still true, then the path to 'Check condition2' will be followed again (loop). If the expression is false, then the method will end.

Code:
The corresponding code looks like this:

```
void methodName( ) {               // method with a repetition
    while ( checkCondition1( ) ){  // check the conditional exp. in diamond
                                   // if the conditional exp. is true, then..
           if ( checkCondition2( )){ // also check cond. exp. in 2nd diamond
                                   // if the 2nd cond. exp. is also true, then..
                step2 ( );         // call the method in the rectangle

           } else {                // if the 2nd cond. exp. is not true, then..
                step1 ( );         // call the method in the rectangle
           }
      }
}
```

The code explained:

- First the value of `checkCondition1()` is determined.
- If the conditional statement in the **`while`** is **`false`** (thus `checkCondition1() == false`), then the method ends.
- If the conditional statement in the **`while`** is **`true`** (thus `checkCondition1() == true`), then the conditional statement behind the **`if`** is evaluated (thus `checkCondition2()`).
  - If the conditional statement following the **`if`** is **`true`** (thus `checkCondition2() == true`), then the code between the curly brackets { and } is executed (thus `step2( )`).
  - If the conditional statement following the **`if`** is **`false`** (thus `checkCondition2() == false`), then the code following the **`else`** is executed (thus `step1( )`).
- After executing `step1( )` or `step2( )`, in both cases the program jumps back to the **`while`** and the conditional statement in the **`while`** is evaluated again (thus `checkCondition1()`). If this is still **`true`**, we repeat the last few steps and the code between the curly brackets is executed again (loop). Otherwise, the method ends.

Note: a sequence, selection or repetition can be used in any order. Any one of them can also be used within any other one.

5 Exercises

5.1 Optimizing

Optimization
A flowchart, and its corresponding code, can sometimes be made more efficient, elegant, or clear without affecting how the program works. Such a simplification is called an *optimization*. An optimization does not affect the operation of the program or final state.

As a result of optimizing the flowchart and code are simplified and often become easier to read, modify (maintain) and test and fewer lines of code are needed. This reduces the chance of errors.

Redundancy
Activities that unnecessarily occur more than once are called *redundant*.

An example of redundancy:

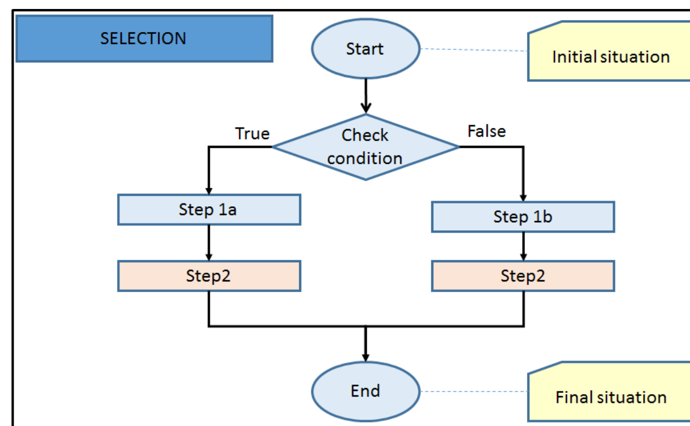


Figure 2: Flowchart with a redundant activity

In the flowchart in figure 2 'Step2' occurs twice.

- If the conditional expression is true, then 'Step 1a' is executed, followed by 'Step2';
- If the conditional expression is false, then 'Step 1b' is executed, followed by 'Step2";
- After that, the method ends.

In both cases 'Step2' is executed as a last step. This flowchart can be simplified by combining the last two steps together. By doing this, 'Step2' will only occur once. This simplification has no effect on how the program works, nor on the final situation. Figure 3 shows the flowchart after this optimization.

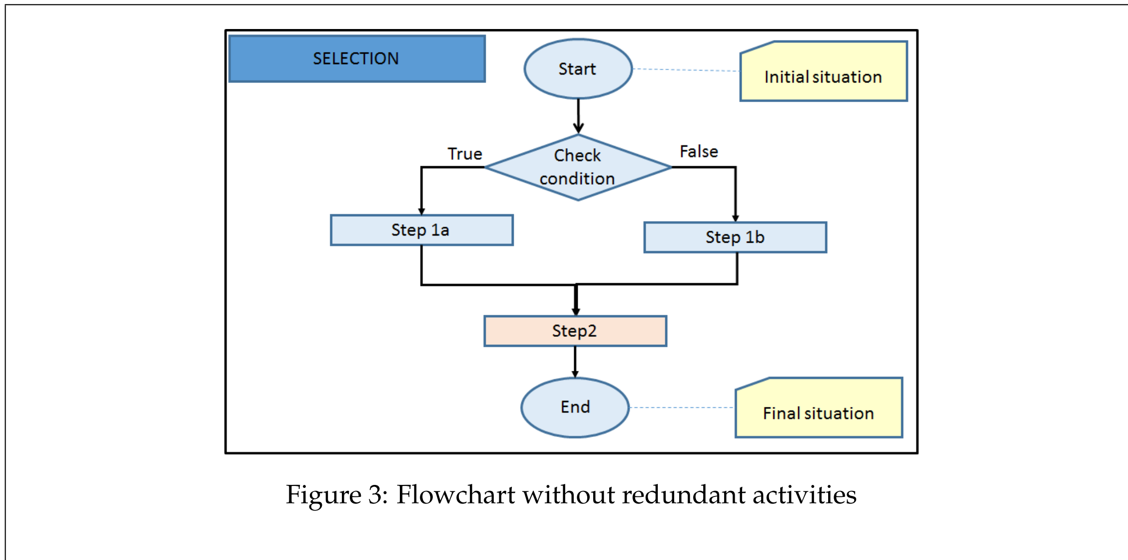


Figure 3: Flowchart without redundant activities

5.1.1 Exercise: removing redundancy

In the following exercises you will practice optimizing redundancy in flowcharts.

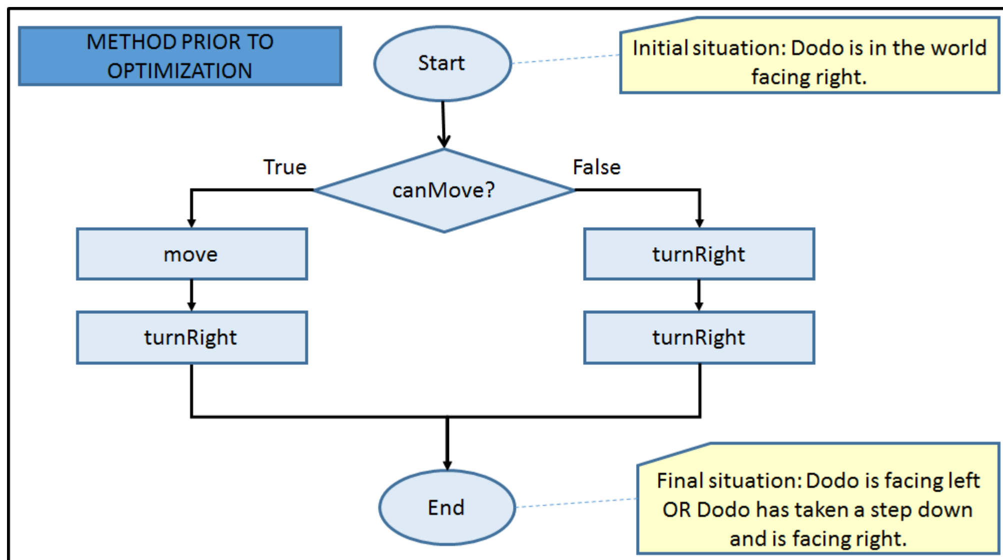


Figure 4: Flowchart prior to optimization

1. Have a look at the flowchart in figure 4.
2. Compare the steps in the left path with those in the right. What do you notice?
3. Optimize the flowchart. Make sure your modification does not affect the operation of the program or its final state.
4. (IN) Have a look at the flowchart in figure 5. Explain why you can't do a similar optimization here.

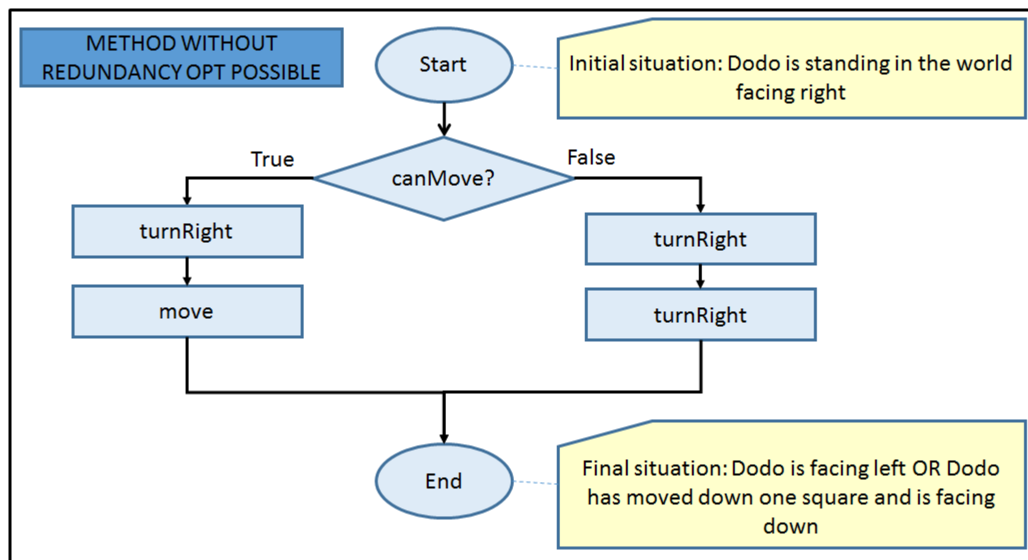


Figure 5: Flowchart where redundancy optimization is not possible

5.2 Sub-methods

Abstraction
A flowchart and its corresponding code can be made clearer by describing a long or complex series of steps separately. These steps are then described in their own sub-flowchart. We refer to the new sub-flowchart from the original flowchart. In code, these steps are written in a *sub-method*. From the method you refer to your new sub-method accordingly. The process of taking details out of the original flowchart or method and describing them separately is called *abstraction*.

Explanation:
Abstraction is used in the following cases:

- repetition: if a particular series of steps repeat;
- many steps: when the flowchart becomes unclear and hard to follow by numerous amount of steps (for example: more than seven).
- cohesive set of instructions: if a set of instructions form a fairly independent cohesive set or module (which may in the future could be re-used elsewhere);
- complex set of instructions: if a set of instructions is so complex that it would be beter (less error-prone) to design, develop and test them separately.

By means of abstraction the code becomes easier to understand, modify and extend. Testing becomes easier too, because the sub-methods can be tested separately. An error (and its cause) can be found quicker. Because code is not repeated unnecessarily, it is less work and futhermore less prone to errors. The sub-method can also be re-used in other methodes or even in other programs (possibly by other programmers). Of course, after testing each sub-method separately, the entire method and program as a whole must be tested.

Example:

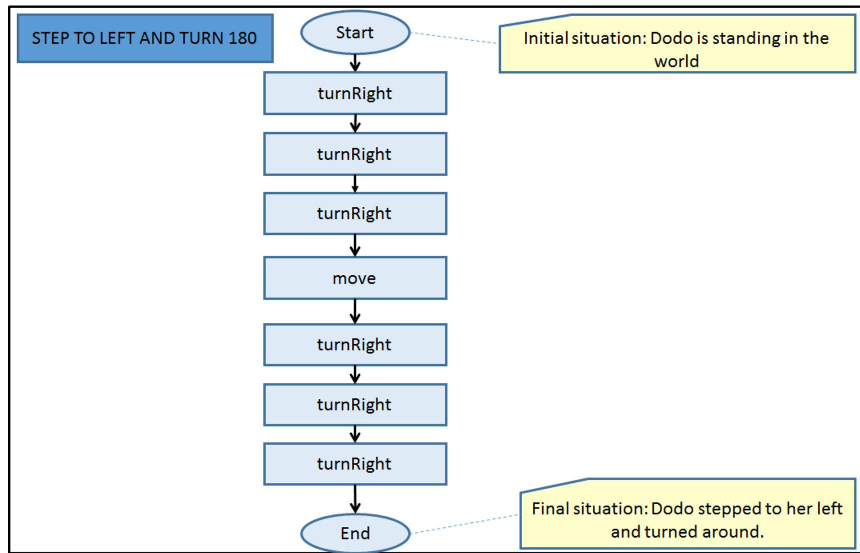


Figure 6: Flowchart prior to optimization

Figure 6 shows a flowchart with a few repeating steps. This flowchart can be simplified by making a separate sub-method (and sub-flowchart) for these steps. The sub-method is then called in the original method. This adjustment has no effect on what the program does or on the final situation. Figure 7 shows what the flowchart looks like after optimization.

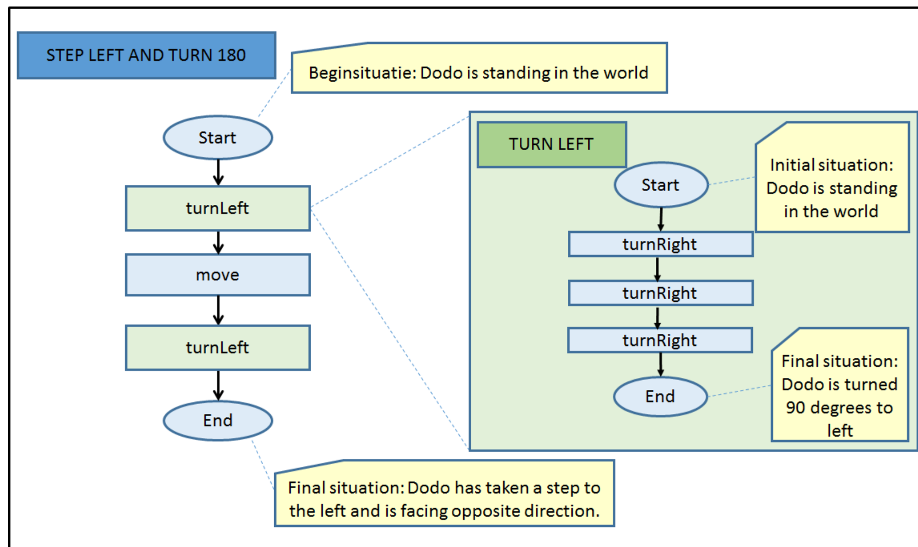


Figure 7: Flowchart after this optimization

5.2.1 Using sub-methods

We are now going to simplify the flowchart from assignment 2 5.3.3 ”Walk around a fence”. We will continue to the use the same code.

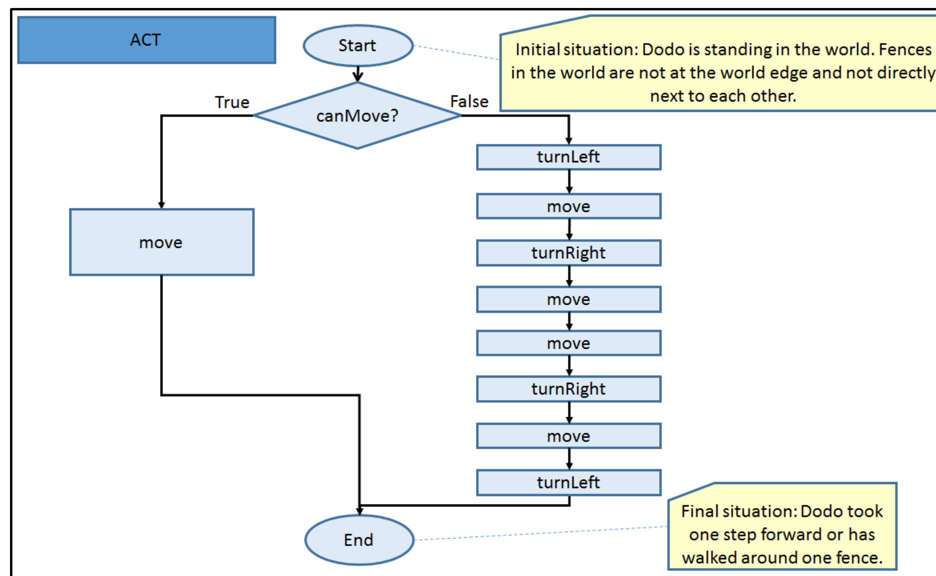


Figure 8: Flowchart prior to optimization

As with every modification, we will do this step-by-step. This reduces the chance of errors. We will follow the steps for modifying code as explained in assignment 2. Let's do this together the first time:

1. (IN) Have a look at the flowchart above. Name the sequence of steps (following the 'False' arrow). Which of the **Flowchart rules** described in assignment 2 are being violated?

2. The first step to using a sub-method is to simplify the flowchart by creating a new sub-flowchart for the long sequence of steps. We will do this in the same manner as in the theory-block above. The goal is to make a sub-method called 'walkAroundFence' for the sequence of steps on the right-hand side of the flowchart:

   (a) We will place these sequence of steps in a (new) sub-flowchart named WALK AROUND FENCE (so, having the same name as the sub-method).

   (b) From the first flowchart we refer to the new sub-diagram (using a blue dotted line).

   (c) The result is shown in figure 9. The new sub-flowchart is shown in green.

   (d) (IN) In the flowchart in figure 8, what should be filled in for A, B, and C?

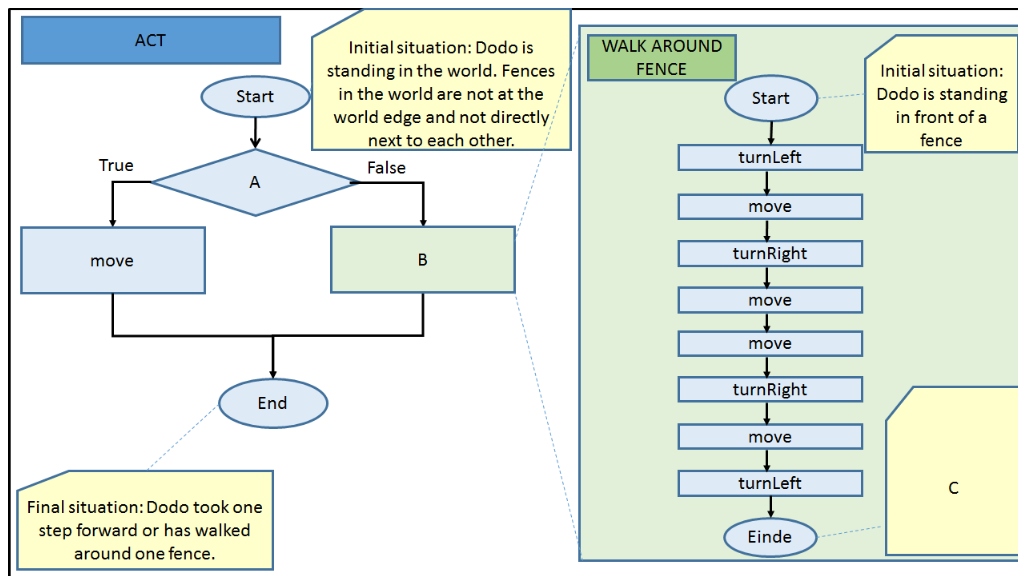


Figure 9: Flowchart `act( )` with a call to a sub-method

3. We will now modify the code so that it corresponds to the new flowchart:

   (a) Open the "world_eggFenceInWay" world. Use **void** `populateFromFile()` to do this.

   (b) Open the `MyDodo` code in the editor.

   (c) We will make a new sub-method for MyDodo called **void** `walkAroundFence()`. This will correspond to the sub-flowchart 'WALK AROUND FENCE'. Copy the following code into `MyDodo`:

```
/**
 *  With this method MyDodo ...
 */
public void walkAroundFence () {


}
```

   (d) Place all the methods shown in the sub-diagram WALK AROUND FENCE in the code in between the curly brackets { and }. Those are exactly the same steps that you determined in assignment 2 5.3.3 'Walk around a fence'.

   (e) Add comments above the sub-method explaining what it does.

   (f) Compile your code.

   (g) Drag Mimi into the world and place a fence in front of her.

   (h) What do you expect Mimi to do if you call **void** `walkAroundFence()`?

   (i) Test your sub-method by right-clicking on Mimi and then selecting **void** `walkAroundFence()`. Does Mimi do exactly what you described in your sub-flowchart?

   (j) Remove the fence in front of Mimi.

   (k) What do you expect Mimi to do now when you call the **void** `walkAroundFence()` sub-method?

   (l) Test the sub-method **void** `walkAroundFence()` using this initial situation. Does Mimi do exactly what is described in the flowchart?

(m) Now that we have tested the sub-method, we have to call it from the `act` method, just like is described in your 'ACT' flowchart (see figure 9). We modify the `act` method so that this calls the new sub-method `walkAroundFence` when Mimi can't move (so when `canMove( )` is false). Change the code in **`void`** `act( )` so that the new sub-method is called after the **`else`**, thus:

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

(n) Compile, run and test your program by pressing the *Act* button. By pressing *Act* you not only test the new `walkAroundFence` sub-method, but the whole act, so also the condition 'canMove' (so Mimi only tries to walk around a fence if she is in front of a fence). Test this with and without a fence in front of Mimi. Does the program work as expected? If not, then follow the steps described in the 'Debugging' theory block in assignment 2.

(o) What do you expect will happen if you call **`void`** `act( )` several times in a row? Check this by pressing the *Run* button.

4. Save your scenario. Later on (in exercise 5.4) we will come back and keep working on this code:

   (a) In Greenfoot choose 'Scenario' from the main menu, and then 'Save As ...'.

   (b) Choose a file name containing your own name and assignment number 3a, for example: `Asgmt3a_John`.

5.3 Run while you can

Run
If you press the *Run* button in Greenfoot, then the whole scenario is executed. That means that for each actor the `act` method is continuously executed. This repeats, for each actor, over and over again until you press the *Pause* button, or until `Greenfoot.stop( )` is called somewhere in the code.

Flowchart:
The flowchart in figure 10 describes the *Run* behaviour.

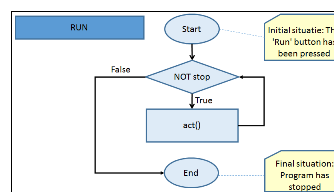


Figure 10: Flowchart for *Run*

The flowchart explained:
When the user presses *Run*, then:

- First the condition expression in the diamond 'NOT stop' is checked;
- If the condition expression is 'False' (that is, the user **has** pressed *Pause* or `Greenfoot.stop( )` has been called in the code), then the *Run* method ends.
- If the conditional expression is 'True', then for each actor the `act` method is called. After the `act` method is executed for each actor, the method jumps back to the diamond and checks the conditional expression again. Is 'NOT stop' still true? Then the 'True' path is followed again and the `act` method is called again (loop). This continues to happen until the condition becomes 'False'. When the condition is 'False', then the method ends.

Note:
In Greenfoot you can execute a method in several ways:

- By calling the method directly. Right-click on the object (for example, Mimi) and select the method.
- Press the *Act* button. For this to work, the method must be called in the `void act( )` code.
- Press the *Run* button. Here too, for this to work, the method must be called in the `void act( )` code. In this case, the method is not called once, but continuously. That can be particularly useful if you want to see what multiple steps do. For example, taking many steps to reach an egg.

5.3.1 (IN) Understanding the `while`

Have a look at the flowchart in figure 10 above.

1. Which language construct (i.e. code control structure, such as `if` .. `then` .. `else` or `while`) is associated with this flowchart?
2. What needs to happen before the 'False' path can be followed?
3. Draw a new flowchart in which you swap 'NOT stop' with 'stop' and 'False' with 'True'. Is the flowchart still correct? Explain why not.

Greenfoot feature: `while` loop in the *Run*
The *Run* command is a built-in feature in Greenfoot and cannot be changed. Therefore, the *Run* flowchart with its `while` loop always remains the same. In a few previous exercises you described algorithms which make use of a `while` to repeat certain steps. An example is the 'find-the-egg' algorithm which used: "While not egg found, take a step." Figure 11 shows the corresponding flowchart. If you wish, you can look back at the flowchart and code that you wrote in assignment 2, part 5.3.1.

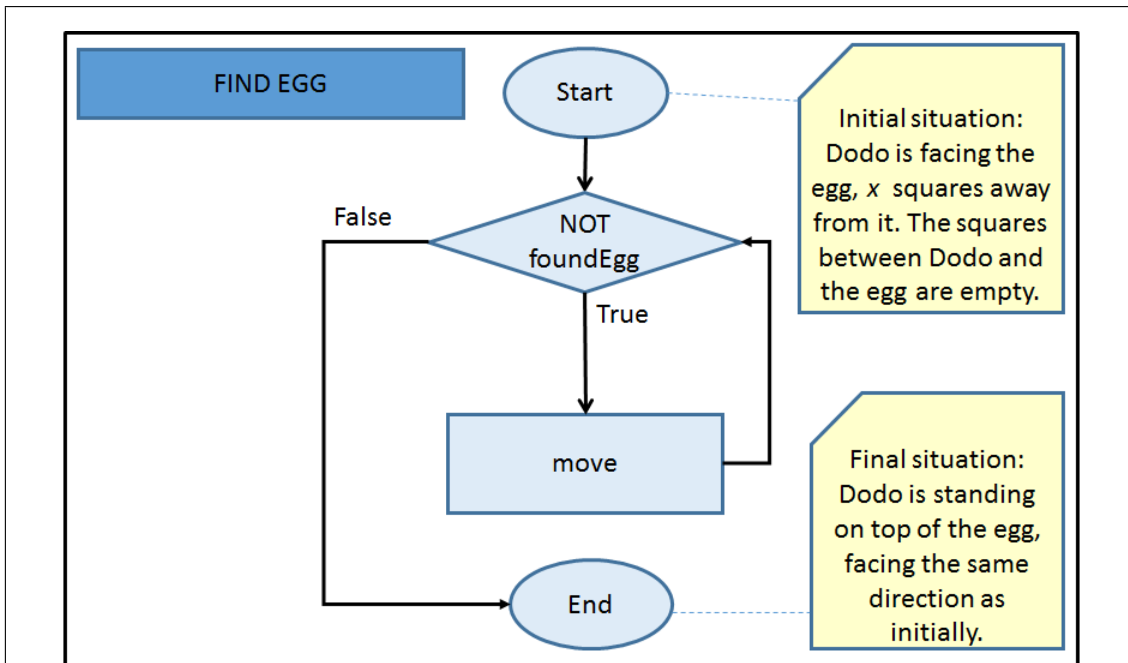
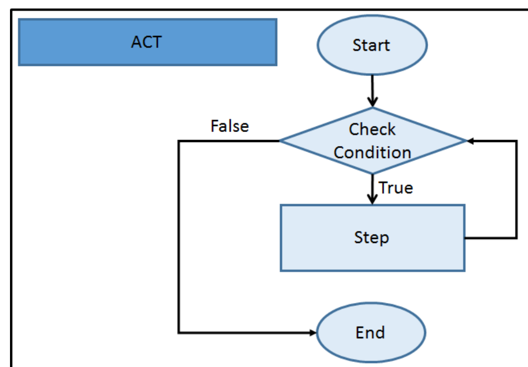


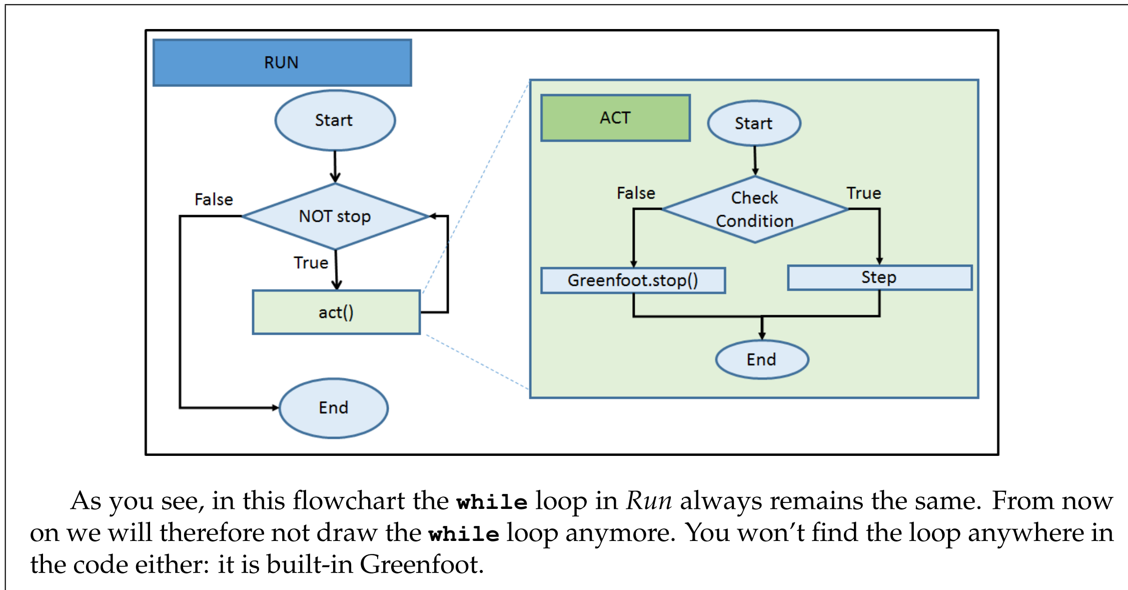
Figure 11: Flowchart for 'find-the-egg' algorithm (assignment 2, part 5.3.1)

If you press the *Run* button, then the `act` method will continuously be executed. That's a part of Greenfoot. But in this case, after executing the `act`, the algorithm will be done. The **`while`** loop in `act` is repeatedly called until the egg has been found. So, after one *Act* the egg will have been found, and in each next `act` call nothing will happen. This makes the use of the *Run* (which repeatedly calls an *Act*) rather useless in this context.

Actually, using the *Run* is only useful if the algorithm consists of repeating the same steps over and over again. The step that must be repeated must then be placed in the `act` method. So, instead of this flowchart:



it's better to draw this flowchart:



As you see, in this flowchart the **while** loop in *Run* always remains the same. From now on we will therefore not draw the **while** loop anymore. You won't find the loop anywhere in the code either: it is built-in Greenfoot.

5.3.2 Find the egg using `while` (instead of `if .. then .. else ..`)

In assignment 2, part 5.3.1 you implemented the 'find-the-egg' algorithm (a summary explanation of the algorithm and its corresponding flowchart is given in the theory block above).

1. In your own words, describe the goal of the algorithm.
2. Have a look at the **void** `act( )` flowchart that you used for that exercise. If you can't find it, then have a look at figure 11 in the theory block above.
3. Which language construct (i.e. code control structure) is associated with this flowchart?
4. Which agreement dit we now make about using a **while** in the `act`?
5. Change the flowchart for the 'find-the-egg' algorithm so that it does *not* use a **while**.
6. We are now going to test if your flowchart without the **while** means the same. To do this we will translate the new flowchart into code and check if the program does what we expect it to do. We're going to modify the code:
   (a) Replace the code in `MyDodo`'s `act` method by the code which originally belonged to the flowchart (with the **while**) to restore the code like it was in assignment 2. That was the following code:

```
/**
 * Keep moving forward until you find the egg
 */
public void act() {
    while( !foundEgg() ){
        move();
    }
}
```

   (b) Compile, run, and test. We are now sure that the original code has been restored properly.
   (c) Now modify this code so that it corresponds to your new flowchart.
   (d) Compile, run, and test.

(e) Open the 'world_Aanroepen6movesAlsWhile.txt' world.

(f) Test your algorithm using the *Run* button. Does the program do what you expect?

(g) What can you say about the agreement that you described in part 4? Does that agreement change the outcome of the program?

7. Save your scenario so that you can hand it in at the end of the assignment.

   (a) In Greenfoot choose 'Scenario' from the main menu, and then 'Save As ...'.

   (b) Choose a file name containing your own name and assignment number 3b, for example: `Asgmt3b_John`.

We have now seen that you can sometimes replace a **`while`** in an `act` method by an **`if`** .. `then` .. **`else`** statement. The *Run* command ensures the repetition by continuously executing the `act` method.

Run or Act?
You maybe wondering "Why on Earth did they add a *Run* button?". There are several reasons for this:

- Some scenarios involve multiple actors, each with their own `act` method. For example: if you place two `MyDodo` instances in the world, then each will have their own `act`. Try it out. What you usually want is that all these objects (more or less) do something at the same time. You won't be able to do that if you have the entire task in the `act` method, but you will if you break the task down into smaller steps.

- You might want the program to respond to *user-input* (mouse or keyboard). Using the *Run* functionality you're not forced to do everything at once in the `act`. By placing everything in the `act`, it may take a long time before the program responds to any user-input. Obviously that's not the intention, and very frustrating for the user. By using the *Run*, your program can quickly respond to user-input.

Run if you can
From now on we will try to set-up the program so that one step in the algorithm is placed in the `act` method. Then, when the *Run* button is pressed, the entire program is executed. For most programs, this is not an issue. It only gets tricky when you want actors to do something quite complex. In that case, we won't use the *Run* or `act`, but rather implement our algorithm in one or more sub-methods and then call this by right-clicking on the object.

5.4 Finding the egg with several obstacles

We will now further investigate how the *Run* or `act` work. Have a look at the following world:

Figure 12: Scenario for exercise 5.4

1. We will continue with the scenario which you saved previously in exercise 5.2.1 part 4 (for example: `Asgmt3a_yourName`). Open that scenario.
2. Open the world: 'world_egg2FenceInWay'.
   (a) Right-click on an empty square in the world.
   (b) Choose **`void`** `populateFromFile()`.
   (c) Navigate to the folder 'worlds'.
   (d) Choose 'world_egg2FenceInWay.txt'
3. (IN) Is your code generic enough to work properly in this world? Can you use it to help Mimi find her egg?
4. Test your code by pressing the *Act* button twice. Does Mimi walk around both fences? If she doesn't, modify your code appropriately.
5. Open the world 'world_egg2FenceInWay' again.
6. What happens now when you press *Run*?
7. (IN) In your own words describe why Mimi doesn't stop after having found the egg. Tip: if you have problems with this, then read the explanation in chapter 5.3 again.

We have now seen that *Act* is repeatedly called by pressing on the *Run* button.

5.5 Giving compliments

`String`
In Java, text is called a `String`. `String` is a *type*, just like **`int`** and **`boolean`**. A parameter can be a `String`. Thus a method call can also be given a `String` (or text) as a parameter. If you want to use a particular text in your program, you need to use quotation marks (see the following example). You can do all sorts of things with `String`s, such as glue two together (called *concatenate*) using '+'. As such, `"Hello"+ " Mimi"` becomes `"Hello Mimi"`.
String example:
The method `showCompliment(String compliment)` has a `String` parameter called 'compliment'. It makes a dialog window appear showing the text 'compliment'.

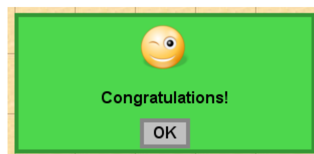


Figure 13: Result after calling:`showCompliment( "Congratulations!");`

When Mimi finds her egg, we want to give her a compliment. To do this, we will add code to the `act( )` method in `MyDodo`.

1. As a starting point we take the flowchart that you made for the `act` in exercise 5.2.1. When Mimi finds the egg, we compliment her on her job well-done. Add this to the flowchart in figure 14.

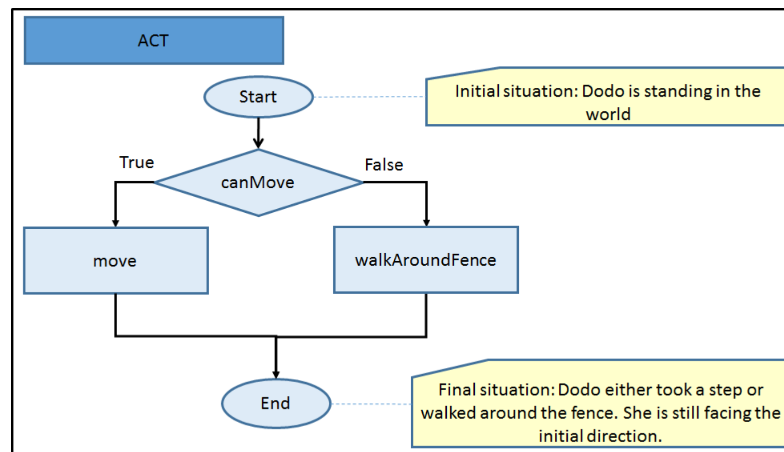


Figure 14: Flowchart for `act` method in "Walk around the fence" exercise (ex 5.2.1)

2. To show the compliment, you can use the method `showCompliment( String compliment )`. This makes a dialog window appear showing the compliment's text.
3. Modify the `act( )` method's code so that a compliment is shown.
4. Don't forget to change the comments above the `act( )` method too.
5. Compile, run, and test the program. Does it work as expected?
6. Change the compliment's text so that Mimi knows why you are congratulating her.
7. Compile and test the program again. Does it work as expected?

We have now seen how you can use a conditional expression to select different methods for execution: using the **`if`** .. `then` .. **`else`** statement, either the methods in the **`if`** branch or the methods in the **`else`** branch will be called. We also saw how to display a dialog window with a particular text.

5.6 Stopping the program

When you press the *Run* button you will see that Mimi doesn't stop after she finds the egg. That's because of Greenfoot's built-in **`while`** loop in the *Run*. We saw this in exercise 5.3. We are now going to modify the program so that it stops when Mimi finds her egg:

1. Open `MyDodo`'s code and find the `act( )` method.
2. Add code to stop the program from running when the egg is found. Tip: use `Greenfoot.stop( );`.
3. Compile and test the program using the *Run* button. Does it work as expected?

We have now seen how you can use `Greenfoot.stop( )` to interrupt the *Run* and stop the program.

5.7 Testing a program in several worlds

In this assignment we've made quite some code modifications. It's time to look back a moment and reflect on what we've done. What can Mimi do well now, and what does she still have to learn?

1. What is the goal of your program? Describe what it must do to work correctly. When are you satisfied? What is the desired final situation?
2. Open the 'world_egg3FenceInWayWithSpace' world.
3. Test the program using *Run*.
4. Explain how it's possible that Mimi finds her egg in this world too.
5. Is your code generic enough to work in the following world too?

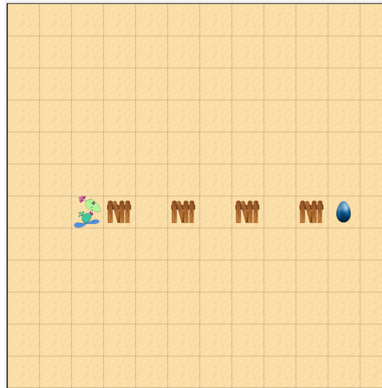


Figure 15: Scenario with four fences

6. (IN) Briefly describe how generic your program is.
7. Drag Mimi, the fences, and the blue egg to different coordinates in the world. Test if the program works as expected. Try a few more positions.
8. (IN) For which initial situations does the program not work correctly?
9. (IN) Describe the initial situation in which the program works correctly.
10. (IN) Make at least two suggestions for improving your program (you don't have to make the improvements, only give suggestions).
11. (IN) What are the two most important things that you have learned during this assignment?

We have now seen that a generic program works in many different situations.

6 Summary

In this assignment you have been introduced to generic and elegant algorithms.

You can now:

- describe how the *Act* and the *Run* works;
- use combinaties of sequences, selections (choices), and repetitions in flowcharts and code;
- write sub-methods for repetitive tasks or to break down complex methods into manageable tasks;
- optimize flowcharts and code;
- modify code in a structured and incremental manner;
- describe the advantages of a generic solution;
- determine if a solution is correct.

7 Saving your work

You have just finished the third 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 ...'.

8 Handing in

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

Hand your (digital) work in via email to renske.weeda@gmail.com:

1. Go to the folder where you saved your work (for example: `Asgmt3b_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. Do the same for the first scenario in this exercise (for example: `Asgmt3a_yourName`).
4. 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.
5. 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'.
6. Hand the compressed (zipped) folder in via Magister.

Make sure you hand your work in before next Wednesday 8:30 (in the morning).