Assignment 2: Designing Dodo's moves

– Algorithmic Thinking and Structured Programming (in Greenfoot) –

1 Introduction

In the previous assignment you were introduced to Greenfoot. Now that you have practiced with the basics of Greenfoot and are able to read some code, you can start writing your own code using some Java *language constructs*.

2 Learning objectives

After completing this assignment, you will be able to:

- decompose a (complex) problem into **subproblems**;
- explain what role **sub-methods** play in solving a problem;
- name the components of a **flowchart**;
- name the steps for drawing a flowchart;
- describe the rules to which a flowchart must comply;
- list quality criteria for flowcharts;
- list optimisation criteria for flowcharts;
- in your own words describe what **modularization** and **abstraction** mean;
- describe the initial and final situations for a problem;
- explain how a flowchart can be used to detect algorithm errors and improvements at an early stage;
- prior to implementation, determine if the steps of an algorithm and its corresponding flowchart will lead to the required solution;
- reason about the **correctness** of a method in terms of the **initial and final situations**;
- recognize a **return statement** in both a flowchart and in code;
- explain why the initial and final situations of an accessor method are identical;
- describe the relationship between a flowchart and code;
- devise an **algorithm** as a solution to a problem;
- identify required **sequences**, **decisions** (or selections) and **repetitions** needed for a solution;
- apply a condition composed of AND, NOT and `boolean` methods;
- visualize an algorithm as a combination of a sequence, decision or a repetition in a flowchart;
- transform a flowchart into code;

- make code modifications in a **structured** and **incremental** manner;
- write, compile, run, and test code;
- name the steps for **debugging** (analysing and tracing errors in code);
- explain, in your own words, what a generic algorithm is;
- compose and implement a **generic algorithm**.

3 Instructions

For this assignment you will need a new scenario:

- Download the **'DodoScenario2'** scenario from Magister.
- Open the scenario.
- 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: `Asgmt2_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.

4 Theory

Flowchart

An *algorithm* can be visualized in a *flowchart*. Subsequently, this can be transformed into code.

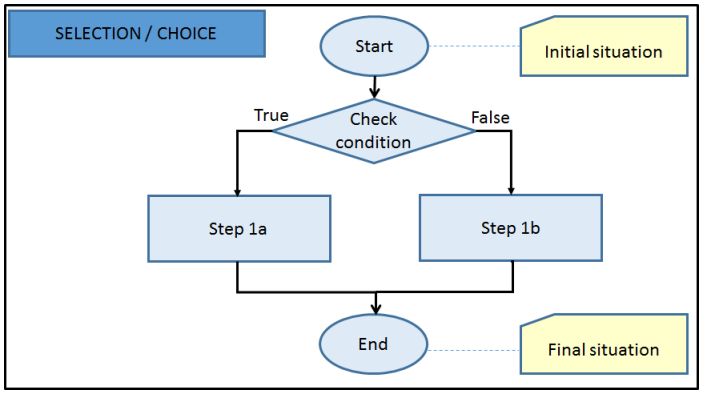

Figure 1: Flowchart

Decomposing a probleem

Sometimes a problem is very complicated and its solution consists of may steps. If you want to solve such a complex problem, you may not know where to start or end up getting lost in all the details. In both cases the problem may seem bigger than it actually is.

A flowchart can help break down (or *decompose*) a big problem into smaller subproblems. First you have to think up a high-level roadmap (or plan) of what has to be done, and in what order. By placing those steps in a flowchart you can visualize the solution as a whole. Now, each of the subproblems can be tackled one-by-one, without having to worry about the bigger picture. After solving a particular subproblem and testing the solution, you do not have to worry about its details anymore. You can use the solution as a building-block in bigger or other problems. This approach, of breaking down a problem, is called *divide-and-conquer*. Of course, when you are done solving all the subproblems, you must check whether you have solved the problem as a whole.

What does a flowchart look like?
A flowchart has the following components:

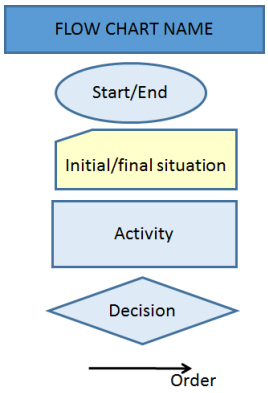

Figure 2: Flowchart components

Flowchart rules

- Each flowchart has a name.
- There are no loose parts 'floating around' (except the diagram's name). All components are connected by arrows or lines (indicating order).
- There is one initial starting point and one final endpoint, each with a description of its situation:
  - Only in exceptional circumstances should there be multiple endpoints.
  - An accessor method which yields a result (i.e. gives an answer to a question), has equal initial and final situations (this will be explained later).
- An activity (rectangle) always has one incoming and one outgoing arrow.
- An decision (diamond) has at least one incoming and at least two outgoing arrows. The outgoing arrows are labelled with their corresponding conditions.
- The flowchart consists of up to seven activities.

Steps for drawing a flowchart
To solve a problem using a flowchart, follow the next steps:

1. **Initial situation**: Briefly describe (in no more than a few words) what the problem/situation is that is to be solved.
2. **Final situation**: When is your problem solved? How do you know that? Describe that in no more than a few words.
3. **Solution strategy**: Decide on how you want to resolve the issue.
4. **Break down** the problem into sub-problems, each of which can further be broken down into smaller sub-subproblems. Continue to do this until each individual problem is small enough to easily solve. Each sub-problem should consist of at most seven steps/-subproblems.
5. For each step or sub-problem:
   - Choose a suitable name (meaningful, consisting of verbs, and formulated at a command).
   - Briefly describe the step.
   - For each step draw a rectangle and use an arrow to connect this to a previous activity or decision.
   - *Modularization*: Determine whether this step should be described in more detail, and thus itself is a step or a problem which must be broken down even further.
6. Draw the **flowchart**.
7. **Check**:
   - **Rules**: Check if the flowcharts adheres to the 'Flowchart rules' (listed above).
   - **Quality**: Check if the flowchart can be simplified or made more elegant (check the 'Flowchart quality criteria' below). For example, is there a sequence of steps which occur repeatedly? Give those steps a name and describe them as a subproblem or sub-method.

Flowchart quality criteria
A flowchart can be used to analyse if your proposed solution is correct or if it can be made more elegant (i.e. smarter). Take the effort to do this before spending time implementing (translating the algorithm into code)! Making adjustments **after** coding costs much more time and effort. Also, it leads to a higher chance of making mistakes.

Using a flowchart you can check if:

- the algorithm is *correct*: it solves the problem;
- initial and final situations have been described (correctly);
- steps follow one another logically;
- it is detailed enough (for unambiguous interpretation);
- the correct choices (*conditions*) are described upon which correct decisions are made;

- there are any exceptions;
- there are no infinite loops (in which case the program never stops, and the computer becomes irresponsive and "freezes");
- there are steps that must can be further decomposed and described in more detail (in a separate flowchart) (*modularization*);
- it is well organized, or whether more use of *abstraction* should be made by moving certain steps to a separate flowchart (*sub-method*). This should be done if the flowchart:
  - consists of more than seven steps;
  - has a sequence of very detailed steps (in that case: *modularize*);
  - certain steps are repeated (in that case: *re-use* by calling sub-methods);
- it can be optimized (made more elegant, efficient, shorter):
  - the same can be done using less steps;
  - is can be made neater and clearer (simpler);
  - components are unnecessary or steps are never executed (for example: a path which will never be reached due to a condition);
  - a complexity improvement (in terms of efficiency) can be made.

The later you make an improvement or resolve an error, the more effort and time it will cost to do. So, first you want to be sure that your algorithm is correct, efficient, reliable and flexible (or 'elegant'). Only after you are sure of that, you should start writing code.

5 Getting started with the exercises

In the following exercises you will develop your own algorithms and try them out in Greenfoot. We will practice working in a structured manner, by first visualising your algorithm in a flowchart, and then translating this into working code, step-by-step.

Steps for creating code

1. Come up with a global plan (or roadmap) for solving the problem.
2. Draw a **flowchart** for the solution. Have a look back at "Steps for drawing a flowchart" in chapter 4.
3. Translate the flowchart into **code**. Pay attention to the naming conventions (discussed in assignment 1).
4. Add **comments** to the code.
5. **Compile** and **Run**. Check if the program does what you expect it to do.
6. **Test** the method by dragging an object into the world, right-clicking on the object, and selecting the method. Test several different situations. Also test using the *Act* button.
7. **Debug**. Repair errors. Try to locate exactly where an error occurs. Make sure the code does exactly what the flowchart specifies. If not, adjust your code.
   If the code corresponds exactly with your flowchart, but the program doesn't do what

you expect, then you may have a mistake in your flowchart. Analyze your flowchart to determine where you've made any incorrect assumption. Return to step 2, modify your flowchart, and follow the proceeding steps again.

8. **Evaluate the solution and reflect on the process**. Has the problem been solved? Which improvements can you suggest? In the process of reaching the solution, what went right? What could have been done better?

Accessor method
An accessor method provides information about an object's state. For example, its result can be an **`int`** (whole number), **`boolean`** ('true' or 'false'), or `String` (text).

Flowchart:
The following is a flowchart corresponding to a **`boolean`** accessor method:

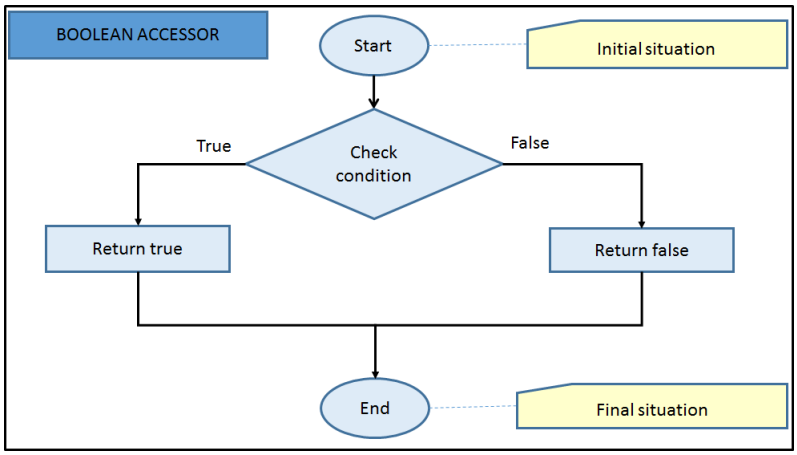

Figure 3: Flowchart for a **`boolean`** accessor method

The flowchart explained:

- First the conditional expression 'Check condition' in the diamond is checked.
- If the conditional expression is true, then the 'True' arrow on the left will be followed and the result **`true`** will be returned.
- If the conditional expression is not true, then the 'False' arrow on the right will be followed and the result **`false`** will be returned.
- After returning a result, a method always terminates (is done). A *return* is always immediately followed by the *End*.
- In an accessor method, the initial and final situation are always equal to each other.

Code:
The corresponding code looks like this:

```
boolean methodName( ) {          // a boolean accessor method
      if( checkCondition( ) ){ // check the conditional expression in the diamond
```

```
                                //if the conditional expression is true
            return true;        // give the result 'true'

        } else {                // if the conditional expression is not true
            return false;       // give the result 'false'
        }
}
```

The code explained:

- First the value of the conditional expression `checkCondition()` is determined.
- If the conditional expression is true, then **`true`** is returned. After that, the method terminates.
- If the conditional expression is not true, then you jump to **`else`**. Here **`false`** is returned. After that, the method terminates.
- After returning a value, nothing more happens. If you try to execute any code after a return, the compiler will complain with the following error message: "unreachable statement";

Note: An accessor method provides information about an object. It should not change the situation. Therefore, from now on we agree that the initial and final situations of an accessor method are equal to each other.

5.1 Exercises: Mimi lost her egg

In the following exercises you will use the scenario: **'DodoScenario2'**.

5.1.1 Sequence of instructions

Have a look at figure 4. Our MyDodo, Mimi, has lost her egg. Can you help her find it?

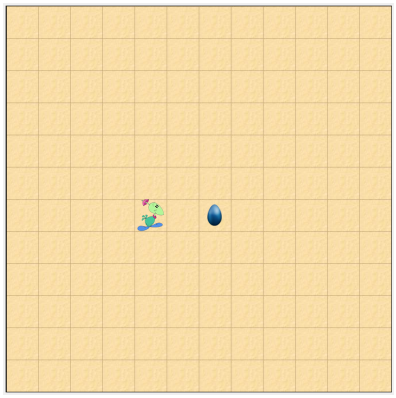

Figure 4: First scenario

Sequence
In a *sequence*, the indicated steps are performed sequentially, one after the other.

Flowchart:
The following is a flowchart for a sequence of steps:

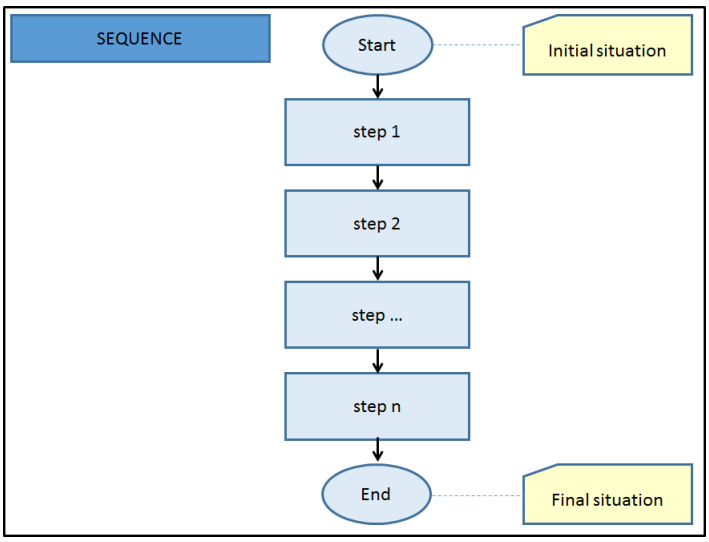

Figure 5: Flowchart for a sequence of steps

Code:
The corresponding code looks like this:

```
void methodName ( ) {   // method with a sequence
      step1  ( );        // call the method in the first rectangle
      step2  ( );        // call the method in the second rectangle
      step...( );        // call the method in the next rectangle
      stepN  ( );        // call the method in the n-th rectangle
}
```

1. Right-click on Mimi. Have a look at all the things she can do (which methods does she have).
2. We will determine what Mimi has to do to reach her egg (i.e. until she is sitting on top of it):
   (a) Come up with a strategy for Mimi.
   (b) Write down which methods you have to call to solve her problem.
   (c) Describe the initial and final situations.
3. (IN) Such a strategy can be described using a flowchart. See figure 6. Fill in the missing parts A, B, and C.

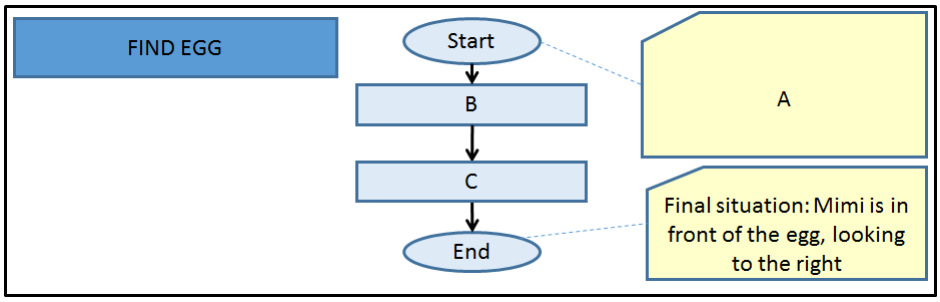

Figure 6: Flowchart for Find Egg

4. Modify the code in MyDodo's `act( )` method so that Mimi follows the steps in your flowchart. Tip: If you need help, then follow the next steps:
   - Right-click on MyDodo in the class diagram (on the right-hand side of the screen).
   - Select 'Open editor'.
   - Find the code for the `act( )` method. Tip: CTRL+F.
   - You have to write your code between the curly brackets { and }.
   - The two rectangles in the flowchart describe the two lines of code you must add. For each rectangle, type the text in the `act( )` method. It will look similar to this:

```
public void act ( ) {
    CALL THE METHOD IN THE FIRST RECTANGLE OF THE FLOWCHART
    CALL THE METHOD IN THE SECOND RECTANGLE OF THE FLOWCHART
}
```

   - Replace the two lines of of capital letters so that it becomes real code. For example: to call 'move', you must type `move ();`
5. Compile (with the *Compile* button) and fix any errors.
6. Press the *Act* button at the bottom of the screen.
7. Test if your program works as expected. Does it reach the final situation as described in your flowchart? If the program does not do what you expect, then look below for some tips on debugging.

Debugging
Test that your program after each minor modification to check if it works as expected. Does it not do what you expect? Then you must retrace your steps in reverse order:

1. Make sure your code matches your flowchart.
2. Check that your flowchart corresponds to the steps (or method calls) which you came up with.
3. Check if the steps you came up are correct and lead to a solution for the problem.

Make it a habit to do this immediately after **each** modification. This will help you find any mistakes much faster.

5.1.2 More instructions in a sequence

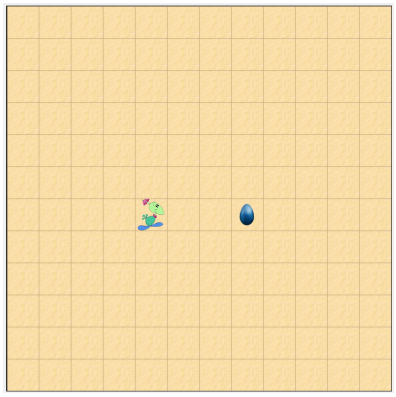

Figure 7: Scenario

1. We will continue with the previous scenario. Adjust the world so that it matches figure 7. Can you help Mimi find her egg again?
2. Richt-click on Mimi.
3. Which of MyDodo's method(s) will you call to get Mimi to her egg?
4. (IN) Draw the corresponding flowchart.
5. Modify the code in MyDodo's `act( )` method.
6. Also change the comments.
7. Compile and test the program.

5.2 The `if .. then .. else`

Is equal to
The *comparison operator* '==' checks if two values are equal to each other.

An example:
Using '`a == 4`' you can compare if '`a`' is equal to '4'. The result is either **`true`** or **`false`**.

Note: The equals-sign '=' (which you use in mathematics) has a different meaning! In code, using '`a = 4`' means 'a becomes 4'.

Selections (choices)
If a conditional expression is true, then do something. Or else, do something else.
For this, you can use an **`if`** .. `then` .. **`else`** statement.

Flowchart:
The following flowchart can be used for a selection (choice):

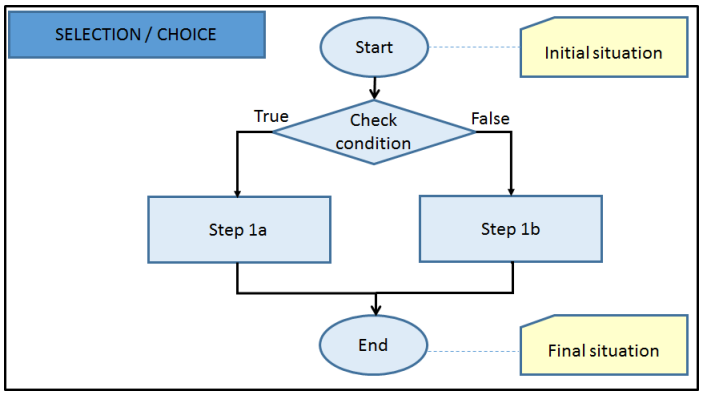

Figure 8: Flowchart with a selection(choice) using an **`if`** .. `then` .. **`else`** statement

The flowchart explained:

- First the conditional expression 'Check condition' in the diamond is checked.
- If the conditional expression is true, then the 'True' arrow on the left will be followed and 'step1a' is executed.
- If the conditional expression is not true, then the 'False' arrow on the right will be followed and 'step1b' is executed.
- After that, the method is done.

Code:
The corresponding code looks like this:

```
void methodName( ) {      // method with a selection
                          // check the conditional expression in the diamond
      if ( checkCondition( ) ) {
                          // if the conditional expression is true
            step1a ( );   // call method in rectangle following the 'True' arrow
      } else {            // if the conditional expression is not true
            step1b ( );   // call method in rectangle following the 'False' arrow
      }
}
```

The code explained:

- First the value of the conditional expression `checkCondition ( )` is determined.
- If the conditional expression is **`true`** (thus `checkCondition( )` == **`true`**), then the code between the curly brackets { and } is executed (thus, `step1a( )`). After that, the method is done.
- If the conditional expression is **`false`** (thus `check( )` == **`false`**), then the code between the curly brackets { and } after the **`else`** is executed (thus, `step1b( )`). After that, the method is done.

Note:

- If nothing needs to happen if the conditional expression is false, then you can just omit the **`else`**-branch.

- Often a "NOT" (negation) is used in conditional expressions (in code: '!'). For example, "NOT borderAhead". This is consistent with the way in which you would describe the algorithm in the words: "if something is NOT the case, then ...".

5.2.1 Can't walk through fences

In assignment 1 we saw that Mimi can't step outside of the world. We will now have a look at the method **boolean** `canMove( )` again.

1. Have a look at the following flowchart:

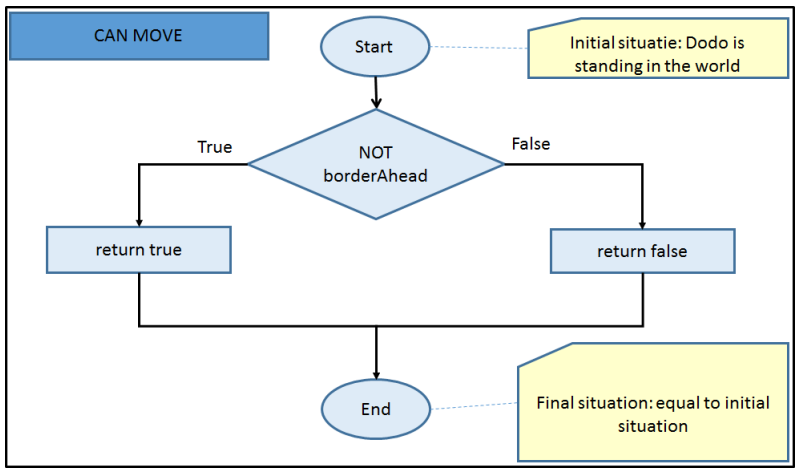

Figure 9: Flowchart `canMove( )` in MyDodo

2. Explain why the description of the final situation is correct.
3. Mimi can't step beyond the world border. However, Mimi doesn't seem to care about the fences. She walks right through them! Place a fence in the world and see for yourself... can she walk through it?
4. Use `Dodo`'s method **boolean** `fenceAhead( )`. What does this do? Try it with a fence in front of Mimi, and again without a fence in front of Mimi.
5. (IN) We don't want Mimi walking through fences (that makes them quite useless). To achieve this we have to modify the **boolean** `canMove( )` method. First we have to decide what to do in the conditional expression.

   Complete the following sentence: Mimi can move if she is NOT in front of a fence AND .....
6. Change the conditional expression in the flowchart (the diamond).
7. Now make the same change in code. Tip: In code 'AND' is written as '&&', also, 'NOT' is written as '!'.
8. Also modify the comments in the code.
9. Compile and test the program. Tip: Does it not work correctly or do you get a compiler-error? Then follow the steps described in the 'Debugging' theory block at the end of chapter 5.1.1.

5.3 The `while` loop

We still have the same mission: "Help Mimi find her egg". Have a look at the scenario in the next picture. How would you solve Mimi's problem?

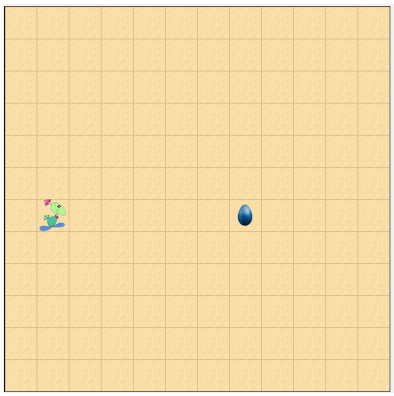

Figure 10: Scenario

Naturally, you could solve this problem in the same way as in the previous exercises, by calling the method `move( )` a certain number of times. However, imagine that Mimi would have to take 1003 steps to get to her egg? As a programmer, you would be very busy typing (or copy-pasting) the `move( );` statement 1003 times. This strategy has a few drawbacks:

- It's a lot of typing or copy-paste work (which is rather boring).
- You might accidently call the `move( )` method 1004 times instead of 1003 times. The result: your program won't work correctly.
- Your program only works for that one specific situation. It is not flexible or general. For example, it will not work if Mimi, in a future scenario, needs 42 steps to get to her egg.

To make your program more general and work in several similar situations, you'll have to devise a smarter *generic* algorithm. What you actually want is a repetition:

"While Mimi has not found her egg, she must take a step."

In the final situation, she'll be done when she finds her egg. Note that this description has no numbers in it.

Generic algorithm
A general algorithm which can be used in multiple initial situations and leads to a correct solution is called *generic*. This does not solve one particular problem, but can be used to solve many similar problems.

Repetition
As long as a particular conditional expression is true, do something.
To achieve this, you can use a repetition, or **`while`** statement.

Flowchart:
The following diagram shows a flowchart with a repetition:

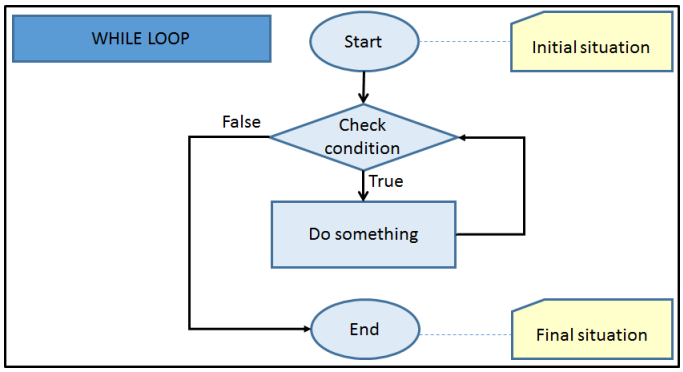

Figure 11: Flowchart for repetition

The flowchart explained:

- First the conditional expression 'Check condition' in the diamond is checked.
- If the conditional expression is not true, then the 'False' arrow is followed to the left and the method done.
- If the conditional expression is true, then the 'True' arrow will be followed downward and 'Do something' is executed.
  Then, we return to the diamond. The conditional expression 'Check condition' in the diamond is checked again. If this conditional expression 'Check condition' is still true, then the path 'True' is followed again, continuously until the conditional expression is 'False'. This is called a loop. Otherwise, the method terminates.

Code:
The corresponding code looks like this:

```
void methodName( ) {          // method with repetition
                              // check the conditional expression in the diamond
      while ( checkCondition( ) ) {
                              // if the conditional expression is true
            doSomething( ); // call the method in the rectangle
      }
}
```

The code explained:

- First the value of the conditional expression `checkCondition ( )` is determined.
- If the conditional expression is **`false`** (thus `checkCondition( ) ==` **`false`**), then the method is done.
- If the conditional expression is **`true`** (thus `checkCondition( ) ==` **`true`**), then the code between the curly brackets { and } after the **`else`** is executed. In this case the method `doSomething( )` is called.

  After that, the conditional expression `checkCondition( )` is checked again. If this conditional expression is still true, then the code between the curly brackets is executed again. These steps are repeated (called a **loop**) until the conditional expression is **`false`**. When the conditional expression is **`false`**, the method ends.

Note:

- Often a "NOT" (negation) is used in conditional expressions (in code: '!'). For example, "NOT egg found". This is consistent with the way in which you would describe the algorithm in the words: "while something is NOT the case, then ...".
- The method `doSomething()` must at some point make the conditional expression become 'false'. At that moment, the repetition stops. A common mistake when using a **`while`** loop is to forget this, the condition remains 'true' and the loop never ends. This is called an infinite loop.

5.3.1 Sequence as a `while`-loop

Help Mimi find her egg. Your solution must be generic. The following initial situation is given:

- Mimi is 0 or more squares away from her egg;
- Mimi is facing in the correct direction (she does not have to turn, she only has to take a certain number of steps forward);
- There is nothing in between Mimi and her egg (for example, there is no fence blocking her way).

We are going to work on this problem step-by-step:

1. We will continue to work with the previous scenario (code). However, we want to open the world in figure 10. To do that, follow the next steps:
   (a) Right-click on the world.
   (b) Choose **`void`** `populateFromFile()`.
   (c) Go to the folder 'worlds'.
   (d) Choose 'world_Aanroepen6movesAlsWhile.txt'.
2. Argue that the following generic algorithm is correct:
   "While Mimi has not found her egg, she should take a step forward." Thus, there is a repetition 'step forward', and a conditional expression "egg not found" (in other words: "NOT egg found").
3. (IN) Have a look at the following flowchart 12. Which step must be repeated? Fill in B.

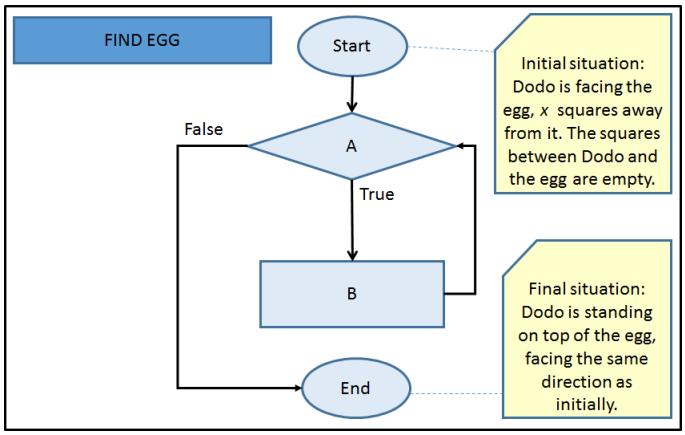

Figure 12: Flowchart "While the egg has **not** been found, take a step forward."

4. Have a look at the conditional expression. Which of Mimi's **boolean** methods could you use to check the conditional expression? Explain in your own words in which cases this method results in **true** or **false**.
5. (IN) In the flowchart, fill in A. Tip: use the **boolean** method from the previous step in combination with 'NOT'.
6. We now add the code to `MyDodo`'s `act( )` method. It should look something like this:

```
public void act ( ) {
    while ( CONDITIONAL EXPRESSION IN DIAMOND ) {
        CALL THE METHOD IN THE RECTANGLE
    }
}
```

7. Replace the code in `act( )` with the above.
8. The conditional statement in the diamond uses 'NOT'. In code this is translated to '!'. So, you would write: `! foundEgg( )`. Replace the text in capital letters and between the brackets '(' and ')' by the correct method call.
9. Also replace the rest of the text in capital letters by one of Mimi's methods. Tip: this is the code corresponding to B in the flowchart.
10. Modify the comments above the `act( )` method accordingly.
11. Compile and test your program. Tip: Does it not work correctly or do you get a compiler-error? Then follow the steps described in the 'Debugging' theory block at the end of chapter 5.1.1.

5.3.2 Walk to the world edge

Write a method that makes Mimi walk to any edge of the world. Her initial position is arbitrary, she can be standing anywhere and facing any direction.

1. We will continue with the previous scenario (the code you have been working on), but will start with an empty world:
   (a) Richt-click on an empty square in the world.
   (b) Choose **void** `populateFromFile( )`.
   (c) Go to the folder 'worlds'.
   (d) Chose 'world_empty.txt'
2. Place Mimi on an arbitrary position in the world.
3. Come up with an algorithm with which Mimi will walk to the edge of the world. The algorithm must be independent of her initial position. Tip: Fill in: "While Mimi is NOT .... Mimi must (do) .... "
4. (IN) Visualise the algorithm in a flowchart.
5. For which initial situations will your algorithm work?
6. (IN) Write a method **void** `walkToEdgeOfWorld( )` by translating your flowchart into code.
7. Use comments to explain your code.
8. Compile and test your method by right-clicking on Mimi. Repeat with Mimi in different positions in the world.

9. Turn Mimi 180 degrees (facing the opposite direction). Does your method also work in the opposite direction? Does your code also work if she is facing any direction? Do your comments properly explain what Mimi can do? If necessary, modify the comments.

5.3.3 (IN) Walk around a fence (writing your own sequence)

Now we're going to teach Mimi something new. If, while she is walking, she runs into an obstacle (for example, a fence), then Mimi should walk around it (over the top). Can you teach her how?

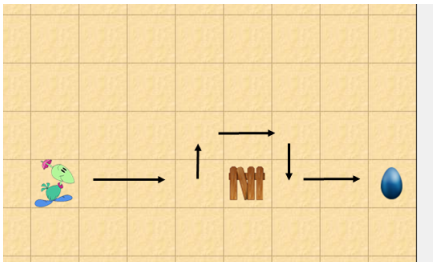

Figure 13: Walk around the fence

We do this as follows:

1. Open the "world_eggFenceInWay" world. Tip: If you don't remember how, have a look at ex. 5.3.1 part 1.
2. While Mimi is walking, if she encounters a fence she will stop. Check this. If Mimi walks right through the fence, then check if you did ex. 5.2.1 correctly before continuing.
3. If Mimi encounters a fence, we want her to turn upwards, take a step, and then continue around the fence, as in figure 13. Complete the following strategy:
   - If Mimi can move forward, take a step forward.
   - Else (so, in this case Mimi can't move forward):
     - turn to the left
     - take a step
     - turn to the ...
     - ...
4. Make sure Mimi is facing to the right again when she is finished.
5. Visualise your strategy in a flowchart.
6. Modify the code in `MyDodo`'s `act( )` method so that it matches your new flowchart.
7. Compile, run and test your program. Tip: Does it not work correctly or do you get a compiler-error? Then follow the 'Debugging' steps described in chapter 5.1.1.
8. Have a look at the description of the initial situation in your flowchart. Does it accurately describe the conditions required for your program to work properly?
9. In this exercise you came up with your own algorithm, visualised it in a flowchart and wrote and tested the code. How did that go? Which things are you proud of? What was easy? What did you have more trouble with? What could you do better next time?

6 Summary

In this assignment you have been introduced to algorithms. In an algorithm you describe how a particular task should be done. You explain each step, choice or repetition very precisely.

You can now:

- come up with a generic solution to a problem;
- devise an algorithm as a sequence of steps, choices (**`if`** .. `then` .. **`else`**) or repetitions (**`while`**);
- write new code in a structured manner by visualising an algorithm in a flowchart and then translating this into code;
- make code modifications incrementally by compiling, running and testing after every minor modification;
- debug step-by-step;
- evaluate the quality of a solution.

6.1 Diagnostic test

1. Assume `MyDodo` had the following method: **`boolean`** `foundAllEggs()`, which indicates whether Mimi has found all of her eggs. Which of the following is true?
   (a) This method has a **`boolean`** parameter.
   (b) The initial and final situations of this method are equal to each other.
   (c) This method has a **`boolean`** as a result.
   (d) This is a mutator method.
2. Name two advantages for using sub-methods.
3. Give two reasons for testing immediately after each minor code change.
4. Explain, in your own words, what the initial and final situations described in the flowchart are useful in programming.
5. Mimi is learning to dance. Draw the flowchart that belongs to the following dance move:
   - Initial situation: Mimi is in the center of the world, facing the left.
   - Dance move: Turn 90 degrees to the right. If Mimi can take a step forward, do that. Or else, turn 90 degrees to the right. Afterwards, in both cases, turn around (180 degrees).
6. Have a look at the flowchart in figure 14. Write the code for this silly dance.

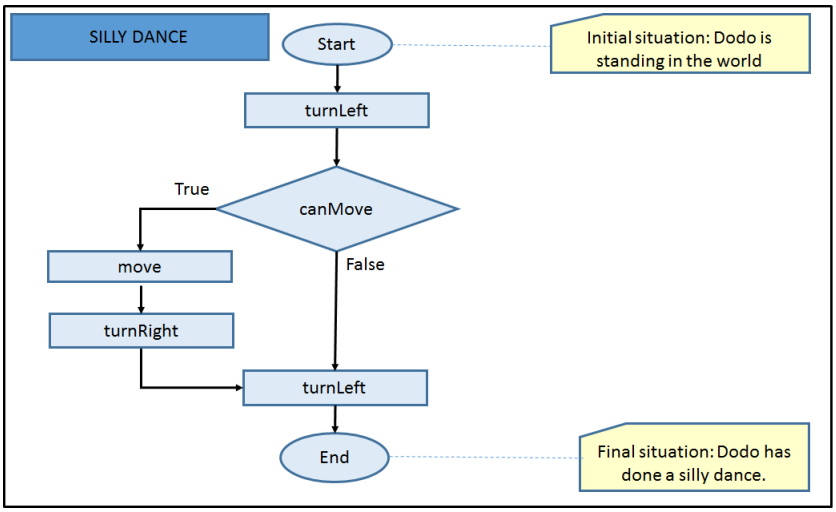

Figure 14: Flowchart for silly dance

6.2 Answers to the diagnostic test

1. Given the method **`boolean`** `foundAllEggs( )`.
   (a) Not true: This method has no parameters.
   (b) True: This method has a result, it returns information about Mimi's state, namely if or if not she has found all her eggs. We agreed that in such a case, the initial and final states must be equal to each other.
   (c) True: This method has a **`boolean`** result, it returns either **`true`** or **`false`**.
   (d) Not true: This is an accessor method.
2. It improves readiblity/understandibility, maintainability, re-use, and testability of the code. It also leads to less redundancy (duplications) in the code.
3. Easier (faster) detection of errors, easier to verify if changes do what they are supposed to do (and don't do what they shouldn't do)
4. The initial situation describes the conditions under which the method will function properly. The final situation describes the expected situation after performing of the method. This way you can easily test if the code does what is expected. It also makes it easier to re-use the code in other parts of the program (or other programs).
5. See figure 15

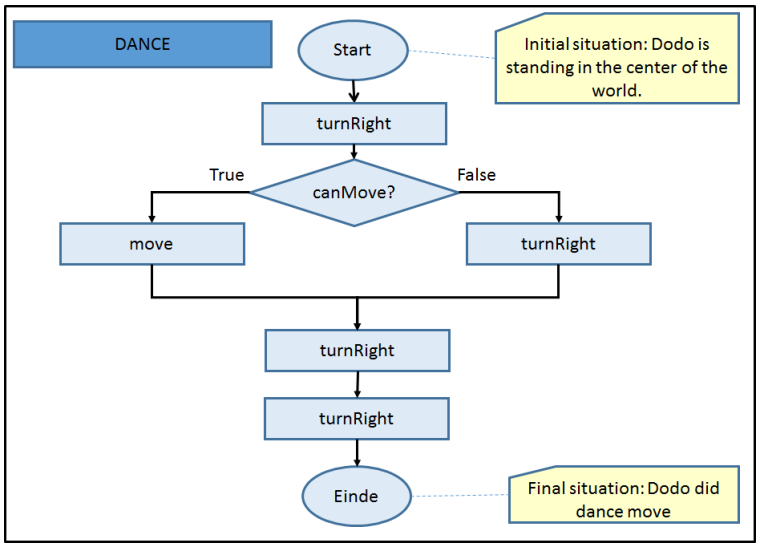

Figure 15: Model answer for diagnostic test question 5

6.
```
/*
* Do a silly dance
*/
public void sillyDance( ) {
    turnLeft( );
    while ( canMove( ) ) {
        move( );
        turnRight( );
    }
    turnLeft( );
}
```

7 Saving your work

You have just finished the second 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. (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.
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 8:30 (in the morning).