are actually dealing with more than one use case. Each of such possible paths should be described in a separate use case.

Example Let us consider an example of the scenario "Correct withdrawal of money ($100) from an ATM with a commission of $5." A use case describing this scenario could look like the one in below.

This use case consists of one main scenario (steps 1–11), three exceptions (labeled 3A, 4A and 9A), and one alternative flow (9B). The numbering of these events refers to the step in which they can occur. If more than one exception or alternate flow is possible in a single step, they are denoted by consecutive letters: A, B, C, etc.

Use case: UC-003-02

Name: successful withdrawal of $100 from an ATM with a commission of $5.

Prerequisites: the user is logged in, the payment card recognized as a card with a withdrawal fee of $5.

Main scenario steps:

1. The user selects the option to *Withdraw money*.
2. The system displays a menu with available payout amounts.
3. The user selects the *$100* option.
4. The system checks if there is at least $105 in the user's account.
5. The system asks for a printout of the confirmation.
6. The user selects the option *NO*.
7. The system displays the message *Remove card*.
8. The system returns the card.
9. The user takes the card.
10. The system withdraws $100 and transfers $5 commission to the bank account.
11. The system displays the message *Take money*. The use case ends, the user is logged out, the system returns to the initial screen.

Final conditions: $100 was withdrawn from the user, the user's account balance reduced by $105, the ATM cash balance reduced by $100.

Alternative flows and exceptions:

3A - the user does not select any option for 30 seconds (the system returns the card and returns to the initial screen).

4A - the account balance is less than $105 (message *Insufficient funds*, return the card, log off the user, return to the initial screen and end the use case).

9A - the user does not take the card for 30 seconds (the system "pulls" the card, sends an email notification to the customer and returns to the initial screen).

9B - the user inserts the card back after receiving it (the system returns the card).

Deriving Test Cases from a Use Case

There is a minimum of what a tester should do to test a use case. This minimum (understood as full, 100% coverage of the use case) is to design:

- One test case to exercise the main scenario, without any unexpected events
- Enough test cases to exercise each exception and alternative flow

So, in our example, we could design the following five test cases:

TC1: implementation of steps 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11
TC2: implementation of steps 1, 2, 3A
TC3: implementation of steps 1, 2, 3, 4A
TC4: implementation of steps 1, 2, 3, 4, 5, 6, 7, 8, 9A
TC5: implementation of steps 1, 2, 3, 4, 5, 6, 7, 8, 9B, 10, 11

Each exception and alternative flow should be tested in separate test cases to avoid the defect masking. We discussed this phenomenon in detail in Sect. 4.2.1. However, in some situations, e.g., time pressure, it may be allowed to test more than one alternative flows within one test case.

Note that TC5 (alternative flow) allows us to achieve the goal (we reach step 11). TC2, TC3, and TC4 end earlier due to the occurrence of incorrect situations during scenario execution. We can design the test cases from the use case, in the form of scenarios, describing the expected response of the system at each step. For example, the scenario for test case TC3 can look like as in Table 4.12.

Let us note that the consecutive steps (user actions and expected responses) correspond to the consecutive use case steps 1, 2, 3, and 4A. Notice also, that the tester checked the boundary value for the withdrawal amount in this scenario: the whole operation was supposed to reduce the user's account balance by $105, while the declared account balance was $104.99. This is a typical example of *combining*

Table 4.12 Test case built from a use case

Test case TC3: ATM withdrawal with insufficient funds in the account (concerning use case UC-003-02, occurrence of exception 4A).
Pre-conditions:
• User is logged in
• The system is on the main menu
• Withdrawal recognized as charged with a $5 commission (due to the type of card)
• Balance on account: $104.99

Step	Event	Expected result
1	Select the option to *Withdraw money*	The system goes to the menu for selecting amounts
2	Choose option $*100*	The system finds that there are no funds in the account, displays the message *No funds*, returns the card

Postconditions:
• The user's account balance and bank's account balance have not changed
• The ATM did not withdraw any money
• The ATM returned the card to the user
• The system returned to the welcome screen

test techniques to include verification of many different types of conditions in a small number of test cases. Here, in verifying the correct operation of the system in a situation of insufficient funds in the account, we used the boundary value analysis technique, so that the insufficient amount was only 1 cent less than the amount that would have already allowed the correct withdrawal of money.

Coverage
The ISO/IEC 29119 standard does not define a coverage measure for use case testing. However, the old syllabus does provide such a measure, assuming that the coverage items are use case scenario flows. Coverage can therefore be defined as the ratio of the number of verified flow paths to all possible flow paths described in the use case.

For example, in the use case described above, there are five different flow paths: the main path, three flows with exception handling, and one with alternative flow. If the test set contained only test cases TC1 and TC5, the use case coverage with these tests would be 2/5 = 40%.

4.3 White-Box Test Techniques

FL-4.3.1 (K2) Explain statement testing.
FL-4.3.2 (K2) Explain branch testing.
FL-4.3.3 (K2) Explain the value of white-box testing.

White-box testing is based on the *internal structure of the test object*. Most often, white-box testing is associated with component testing, in which the model of the test object is the internal structure of the code, represented, for example, in the form of a so-called control flow graph (CFG). However, it is important to note that white-box test techniques can be applied to all test levels, e.g.:

- At component testing (example structure: CFG)
- At integration testing (example structures: call graph, API)
- At system testing (example structures: business process modeled in BPMN language,[3] program menu)
- At acceptance testing (example structure: website pages structure)

The Foundation Level syllabus discusses two white-box test techniques: statement testing and branch testing. Both of these techniques are by their nature associated with code, so their area of application is mainly component testing. In addition to these, there are many other (and usually more powerful) white-box techniques, such as:

[3] Business Process Model and Notation, BPMN—a graphical notation used to describe business processes

- MC/DC testing
- Multiple conditions testing
- Loop testing
- Basis path testing

However, these techniques are not discussed in the Foundation Level syllabus. Some of them are discussed in the Advanced Level—Technical Test Analyst syllabus [29]. These stronger white-box test techniques are often used when testing safety-critical systems (e.g., medical instrument software or aerospace software).

4.3.1 Statement Testing and Statement Coverage

Statement testing is the simplest and also the weakest of all white-box test techniques. It involves covering *executable statements* in the source code of a program.

Example Consider a simple code snippet:

```
1. INPUT x, y // two natural numbers
2.  IF (x > y) THEN
3.    z := x – y
    ELSE
4.    z := y – x
5.  IF (x > 1) THEN
6.    z := z * 2
7.  RETURN z
```

Executable statements are marked with the numbers 1–7 in this code. The text starting with the "//" characters is a comment and is not executed when the program runs. In general, the code is executed sequentially, line by line, unless some jumps in the control flow occur (e.g., in the decision statements 2 and 5 in our code). The keyword **ELSE** is only part of the syntax of the **IF-THEN-ELSE** statement and by itself is not treated as executable instruction (because there is nothing to execute here).

After taking from the input two variables, x and y, in statement 1, the program checks in statement 2 whether x is greater than y. If so, statement 3 is executed (assignment to z the difference x − y), followed by a jump to statement 5. If not, statement 4 (the "else" part of the **IF-THEN-ELSE**) is executed, in which the difference y − x is assigned to z variable, followed by a jump to statement 5. In statement 5, it is checked whether the x variable has a value greater than 1. If so, the body of the **IF-THEN** statement (statement 6) is executed (doubling the value of z). If the decision in statement 5 is false, statement 6 is skipped, and the control flow jumps after the **IF-THEN** block, that is, to statement 7, where the result—the value of the variable z—is returned and the program terminates its operation.

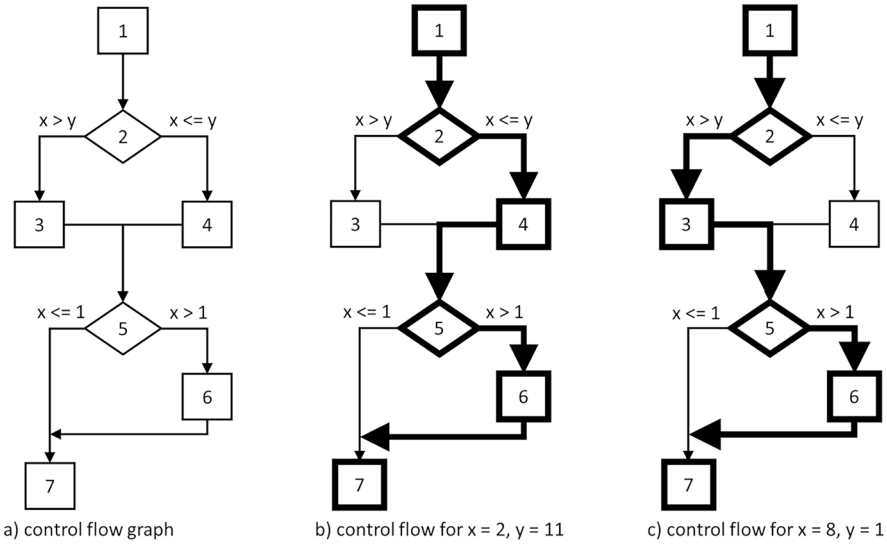

Fig. 4.12 CFG and examples of control flow for different input data

The source code can be represented as a CFG. Such a graph for the abovementioned code is shown in Fig. 4.12a. It is a directed graph, in which the vertices represent statements and the arrows represent the possible control flow between statements. Decision statements are denoted here by rhombuses while other statements by squares.

Consider two test cases for the code from the above example:

- TC1: input: $x = 2$, $y = 11$; expected output: 18.
- TC2: input: $x = 8$, $y = 1$; expected output: 14.

In Fig. 4.12b, the bolded arrows show the control flow when the input is given the values $x = 2$, $y = 11$. The control flow for TC1 will be as follows (in the parentheses, we show the decision outcome at a given decision statement):

$1 \rightarrow 2\ (x \leq y) \rightarrow 4 \rightarrow 5\ (x > 1) \rightarrow 6 \rightarrow 7.$

On the other hand, for the TC2 with input $x = 8$, $y = 1$, the control flow is shown in Fig. 4.12c and will be as follows:

$1 \rightarrow 2\ (x > y) \rightarrow 3 \rightarrow 5\ (x > 1) \rightarrow 6 \rightarrow 7.$

Let us emphasize one very important thing in the context of white-box test techniques: the expected results in test cases are *not* derived from the code. This is because the code is precisely what we are testing—so it cannot be its own oracle. The expected results are derived from a specification external to the code. In our case, it could be like this: "The program takes two natural numbers and then subtracts the

smaller one from the larger one. If the first number is greater than 1, the value is further doubled. The program returns the value calculated in this way."

Coverage
In statement testing, the coverage items are the *executable statements*. Thus, the **statement coverage** 🕮 is the quotient of executable instructions covered by test cases by the number of all executable instructions in the analyzed code, generally expressed as a percentage. Note that this metric does not take into account statements that are not executable (e.g., comments or function headers). Let us reconsider the above source code and test cases TC1 and TC2. If our test set contains only TC1, its execution will achieve a coverage of 6/7 (ca. 86%), since this test case exercises six different executable instructions out of the seven that our code consists of (these are 1, 2, 4, 5, 6, and 7). TC2 also exercises six of the seven statements (1, 2, 3, 5, 6, 7), so it achieves the same coverage of ca. 86%. However, if our test set contains both TC1 and TC2, the coverage will be 100%, because *together*, these two test cases exercise all seven statements (1, 2, 3, 4, 5, 6, 7).

4.3.2 Branch Testing and Branch Coverage

A *branch* is a control flow between two vertices of a CFG. A branch can be unconditional or conditional. An unconditional branch between vertices A and B means that after the execution of statement A is completed, control *must* move to statement B. A conditional branch between A and B means that after the execution of statement A is completed, control *can* move on to statement B, but not necessarily. Conditional branches come out of decision vertices, that is, places in the code where some decision is made on which the further course of control depends. Examples of a decision statement are the **IF-THEN**, **IF-THEN-ELSE**, and **SWITCH-CASE** conditional statements, as well as statements that check the so-called loop condition in **WHILE**, **DO-WHILE**, or **FOR** loops.

For example, in Fig. 4.12a, the unconditional branches are 1→2, 3→5, 4→5, and 6→7, because, for example, if statement 3 is executed, the next statement *must* be statement 5. The other branches, 2→3, 2→4, 5→6, and 5→7, are conditional. For example, after the execution of statement 2, the control can go to statement 3 or to statement 4. Which of these cases occurs will depend on the logical value of the IF decision in statement 2. If $x > y$, the control will go to 3, but if $x \leq y$, it will go to 4.

Coverage
In branch testing, the coverage items are *branches*, both conditional and unconditional. **Branch coverage** 🕮 is therefore calculated as the number of branches that were exercised during test execution divided by the total number of branches in the code. The goal of branch testing is to design a sufficient number of test cases that achieve the required (accepted) level of branch coverage. As in other cases, this coverage is often expressed as a percentage. When full, 100% branch coverage is achieved, it means that every branch in the code—both conditional and

unconditional—was executed during the tests at least once. This means that we have tested all possible direct transitions between statements in the code.

Example Consider again the example code from Sect. 4.3.1:

```
1. INPUT x, y // two natural numbers
2. IF (x > y) THEN
3. z := x - y
   ELSE
4.    z := y - x
5. IF (x > 1) THEN
6.    z := z * 2
7. RETURN z
```

In this code, we have eight branches:

$1 \rightarrow 2$, $2 \rightarrow 3$, $2 \rightarrow 4$, $3 \rightarrow 5$,
$4 \rightarrow 5$, $5 \rightarrow 6$, $5 \rightarrow 7$, $6 \rightarrow 7$.

Test case TC1 (x = 2, y = 11, expected output: 18) covers the following branches:

- $1 \rightarrow 2$ (unconditional)
- $2 \rightarrow 4$ (conditional)
- $4 \rightarrow 5$ (unconditional)
- $5 \rightarrow 6$ (conditional)
- $6 \rightarrow 7$ (unconditional)

and achieves $5/8 = 67.5\%$ coverage.

Test case TC2 (x = 8, y = 1, expected output: 14) covers the following branches:

- $1 \rightarrow 2$ (unconditional)
- $2 \rightarrow 3$ (conditional)
- $3 \rightarrow 5$ (unconditional)
- $5 \rightarrow 6$ (conditional)
- $6 \rightarrow 7$ (unconditional)

and achieves also $5/8 = 67.5\%$ coverage.

The two cases together achieve $7/8 = 87.5\%$ branch coverage, because all branches are covered except one, $5 \rightarrow 7$.

Example Consider the code and its CFG from Fig. 4.13.

A single test case with input y = 3 achieves 100% statement coverage. This is because the control will go through the following statements (the value of the variable x assigned in statement 4 and the decision checked in statement 3 are given in parentheses):

$1 \rightarrow 2\,(x := 1) \rightarrow 3\,(1 < 3) \rightarrow 4\,(x := 2) \rightarrow 3\,(2 < 3) \rightarrow 4\,(x := 3) \rightarrow 3\,(3 < 3) \rightarrow 5.$

```
1.   INPUT y
2.   x=1;
3.   WHILE (x < y) DO
4.      x=x+1;
5.   PRINT ("End of the loop")
     END
```

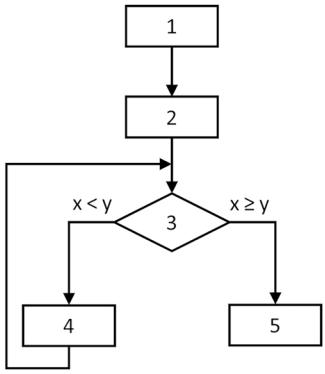

Fig. 4.13 The source code with a while loop and its CFG

The same test case also achieves 100% branch coverage. There are five branches in the code: 1 → 2, 2 → 3, 3 → 4, 3 → 5, and 4 → 3. According to the control flow, branches will be covered sequentially as follows:

1 → 2 (unconditional)
2 → 3 (unconditional)
3 → 4 (conditional)
4 → 3 (unconditional)
3 → 4 (conditional, covered earlier)
4 → 3 (unconditional, covered earlier)
3 → 5 (conditional)

Example Consider the code in Fig. 4.14a.

Its CFG, in which each vertex corresponds to a single statement, is shown in Fig. 4.14b. There are nine branches in this graph:

- 1→2 (unconditional)
- 2→3 (unconditional)
- 3→4 (conditional)
- 3→5 (conditional)
- 5→6 (conditional)
- 5→9 (conditional)
- 6→7 (unconditional)
- 7→8 (unconditional)
- 8→5 (unconditional)

To achieve 100% branch coverage, we need at least two test cases, for example:

TC1: 1→2→3→4
TC2: 1→2→3→5→6→7→8→5→9

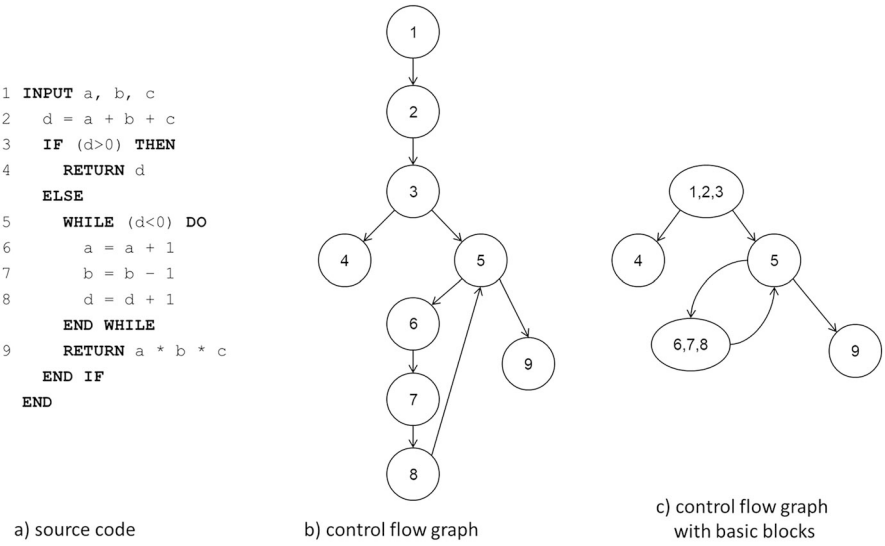

Fig. 4.14 Two CFGs for the same code

TC1 covers three out of nine branches, so it achieves 3/9 ≈ 33.3% branch coverage. TC2 covers eight of the nine branches, so it achieves 8/9 ≈ 89% branch coverage. The two tests together cover all nine branches, so the test set {TC1, TC2} achieves full branch coverage, 9/9 = 100%.

Sometimes, a CFG is drawn so as to include several statements in a single vertex, constituting a so-called basic block. A basic block is a sequence of statements such that when one of them executes, all the others must execute. A CFG modified in this way is smaller and more readable (there are no "long chains of vertices" in it), but its use affects the coverage measure, since there will be fewer edges than in a CFG with no basic blocks. For example, the graph in Fig. 4.14c is equivalent to the graph in Fig. 4.14b but has only five edges. The TC1 mentioned earlier achieves 1/5 = 20% branch coverage in this case, while the TC2 achieves 4/5 = 80% coverage.

If coverage is measured directly on the code, it is equivalent to the coverage calculated for a CFG, in which each vertex represents a single instruction, since branches then represent the direct flow of control between individual instructions, not their groups (basic blocks).

4.3.3 The Value of White-Box Testing

Statement testing is based on the following so-called error hypothesis: if there is a defect in the program, it must be located in one of the executable statements. And if we achieve 100% statement coverage, then we are sure that a statement with a defect

was executed. This, of course, will not guarantee the triggering of a failure, but at least it will create such a possibility. Consider a simple example of code that takes two numbers as input and returns their multiplication (if the first number is positive) or returns the first number (if the first number is not positive). The code looks as follows:

```
1. INPUT x, y
2. IF x > 0 THEN
3.      x := x + y
4. RETURN x
```

In this program, there is a defect involving the incorrect use of the addition operator instead of multiplication in line 3. For the input data x = 2, y = 2, the program will go through all the statements, so full 100% statement coverage will be achieved. In particular, statement 3 will be executed incorrectly, but incidentally, 2 * 2 is equal to 2 + 2, so the returned result will be correct despite the defect. This simple example shows that statement coverage is not a strong technique, and it is worth using other, stronger ones, such as branch coverage testing.

In branch testing, on the other hand, the error hypothesis is as follows: if there is a defect in the program, it causes an erroneous control flow. In a situation where we have achieved 100% branch coverage, we must have exercised at least once a transition that was wrong (e.g., the decision value should be true, but was false), so that the program will execute the wrong statements, and this will possibly result in a failure.

Of course, as in the case of statement testing, full branch coverage does not guarantee that a failure will occur, even if the program takes the wrong path. In our example, when we add a second test case with x = 0, y = 5, we will additionally cover the branch 2 → 3. The two test cases together achieve 100% branch coverage, but still, the defect in the code will not be identified—the actual results in both test cases will be exactly the same, as the expected results.

However, there is an important relationship between statement testing and branch testing:

Achieving 100% branch coverage guarantees the achievement of 100% statement coverage.

In scientific jargon, we say that branch coverage *subsumes* the statement coverage. This means that *any* test set that achieves 100% branch coverage, by definition, also achieves 100% statement coverage. Thus, we don't need to check statement coverage separately in such a case. We know that it must be 100%.

The inverse subsumption relationship is not true—achieving 100% statement coverage does not guarantee that 100% branch coverage is achieved. To demonstrate this, let us consider the code snippet given above. This program has four statements and four branches: 1→2, 2→3, 2→4, and 3→4. For the input data x = 3, y = 1, the program will go through all four statements, 1, 2, 3, and 4, so this test case will achieve 100% statement coverage. However, we have not covered all branches—

after the execution of statement 2, the program goes to statement 3 (because $3 > 0$), so it does not cover branch 2→ 4 (which would be executed in the case where the decision in statement 2 would be false). So the test for x = 3, y = 1 achieves 100% statement coverage but only 75% branch coverage.

Note also that each decision statement is a *statement* and as such is part of the set of executable statements. For example, in the code above, line 2 is a statement (and is treated as such when we use the statement coverage technique). So we can say that 100% statement coverage guarantees the *execution of* every decision in the code, but we can't say that it guarantees the achievement of every *outcome of* every decision (and so we can't say that it guarantees the coverage of all branches in the code).

The key advantage of all white-box test techniques is that the *entire software implementation* is taken into account during testing, making it easier to detect defects even when the software specification is unclear or incomplete. Test design is independent of the specification. A corresponding weakness is that if the software does not implement one or more requirements, white-box testing may fail to detect resulting defects involving lack of functionality [50].

White-box test techniques can be used in static testing. They are well suited for reviews of code that is not yet ready for execution [51], as well as pseudocode and other high-level logic that can be modeled using a control flow diagram.

If only black-box testing is performed, there is no way to measure actual code coverage. White-box testing provides an objective measure of coverage and provides the necessary information to allow additional tests to be generated to increase that coverage and subsequently increase confidence in the code.

4.4 Experience-Based Test Techniques

FL-4.4.1 (K2) Explain error guessing.
FL-4.4.2 (K2) Explain exploratory testing.
FL-4.4.3 (K2) Explain checklist-based testing.

In addition to specification-based and white-box test techniques, there is a third family of techniques: experience-based test techniques. This category contains techniques that are considered less formal than those discussed earlier, where the basis of the tester's actions was always some formal model (a model of the domain, logic, behavior, structure, etc.).

Experience-based test techniques primarily make use of testers' knowledge, skills, intuition, and their experience working with similar products or even the same product before. This approach makes it easier for the tester to identify failures or defects that would be difficult to detect using more structured techniques. Despite appearances, experience-based test techniques are commonly used by testers. However, it is important to keep in mind that the effectiveness of these techniques, by

their very nature, will depend heavily on the approach and experience of individual testers. The Foundation Level syllabus lists the following three types of experience-based test techniques, which we will discuss in the following sections:

- Error guessing
- Exploratory testing
- Checklist-based testing

4.4.1 Error Guessing

Description of the Technique

Error guessing 🕮 is perhaps conceptually the simplest of all experience-based testing techniques. It is not associated with any prior planning, nor is it necessary to manage the tester's activities associated with using this technique. The tester simply predicts the types of errors possibly made by developers, defects, or failures in a component or system by referring to, among other things, the following aspects:

- How the component or system under test (or its previous version already working in production) has worked so far
- What are the typical errors the tester knows, made by developers, architects, analysts, and other members of the production team
- Knowledge of failures that have occurred previously in similar applications tested by the tester or that the tester has heard of

Errors, defects, and failures in general can be related to:

- Input (e.g., valid input not accepted, invalid input accepted, wrong parameter value, missing input parameter)
- Output (e.g., wrong output format, incorrect output, correct output at the wrong time, incomplete output, missing output, grammatical or punctuation errors)
- Logic (e.g., missing cases to consider, duplicate cases to consider, invalid logical operator, missing condition, invalid loop iteration)
- Calculations (e.g., wrong algorithm, ineffective algorithm, missing calculation, invalid operand, invalid operator, bracketing error, insufficient precision of the result, invalid built-in function)
- Interface (e.g., incorrect processing of events from the interface, time-related failures in input/output processing, calling the wrong or non-existent function, missing parameter, incompatible parameter types)
- Data (e.g., incorrect initialization, definition, or declaration of a variable, incorrect access to a variable, wrong value of a variable, use of an invalid variable, incorrect reference to a data, incorrect scaling or unit of a data, wrong dimension of the data, incorrect index, incorrect type of a variable, incorrect range of a variable, "error by one" (see Sect. 4.2.2))

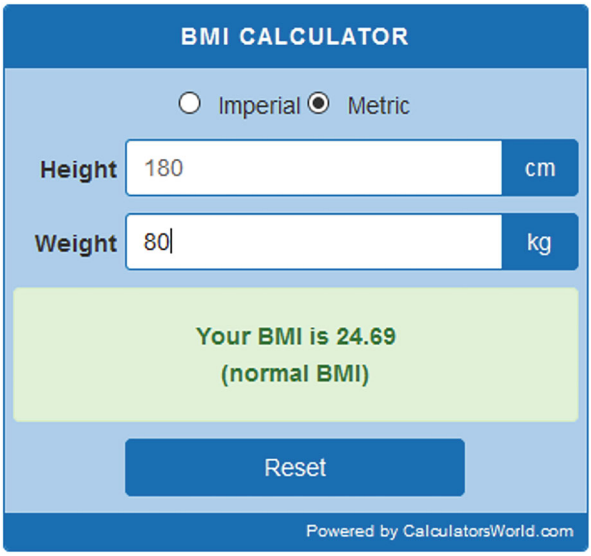

Fig. 4.15 BMI calculator (source: calculatorsworld.com)

There is a more organized, methodical variation of this technique, called *fault attack* or *software attack*. This approach involves creating a list of potential mistakes, defects, and failures. The tester analyzes the list point by point and tries to make each mistake, defect, or failure appropriately visible to the object under testing.

Many examples of such attacks are described in [52–54]. This technique differs from the others in that its starting point is a "negative" event, a specific defect or failure, rather than a "positive" thing to verify (such as the input domain or business logic of the system).

A tester can create lists of errors, defects, and failures based on their own experience—then they will be the most effective. They can also use various types of lists that are publicly available, for example, on the Internet.

Example The tester is testing a program to calculate BMI (Body Mass Index), which takes two values from the user as input, weight and height, and then calculates the BMI. The application's interface is shown in Fig. 4.15.

The tester uses a fault attack using the following list of defects and failures:

1. The occurrence of an overflow for a form field.
2. The occurrence of division by zero.
3. Forcing the occurrence of an invalid value.