5.7 `break` and `continue` Statements

- `break`/`continue` statements
  - Alter flow of control
- `break` statement
  - Causes immediate exit from control structure
  - Used in `while`, `for`, `do…while` or `switch` statements
- `continue` statement
  - Skips remaining statements in loop body
    - Proceeds to increment and condition test in `for` loops
    - Proceeds to condition test in `while`/`do…while` loops
  - Then performs next iteration (if not terminating)
  - Used in `while`, `for` or `do…while` statements

```cpp
// Fig. 5.13: fig05_13.cpp
// break statement exiting a for statement.
#include <iostream>
using std::cout;
using std::endl;

int main()
{
   int count; // control variable also used after loop terminates

   for ( count = 1; count <= 10; count++ ) // loop 10 times
   {
      if ( count == 5 )
         break; // break loop only if x is 5

      cout << count << " ";
   } // end for

   cout << "\nBroke out of loop at count = " << count << endl;
   return 0; // indicate successful termination
} // end main
```

Loop **10** times

Exit **for** statement (with a **break**) when **count** equals **5**

```
1 2 3 4
Broke out of loop at count = 5
```

Outline

fig05_14.cpp

(1 of 1)

```
1  // Fig. 5.14: fig05_14.cpp
2  // continue statement terminating an iteration of a for statement.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  int main()
8  {
9     for ( int count = 1; count <= 10; count++ ) // loop 10 times
10    {
11       if ( count == 5 ) // if count is 5,
12          continue;      // skip remaining code in loop
13
14       cout << count << " ";
15    } // end for
16
17    cout << "\nUsed continue to skip printing 5" << endl;
18    return 0; // indicate successful termination
19 } // end main
```

Loop **10** times

Skip line **14** and proceed to line **9** when **count** equals **5**

```
1 2 3 4 6 7 8 9 10
Used continue to skip printing 5
```

Good Programming Practice 5.12

Some programmers feel that `break` and `continue` violate structured programming. The effects of these statements can be achieved by structured programming techniques we soon will learn, so these programmers do not use `break` and `continue`. Most programmers consider the use of `break` in `switch` statements acceptable.

Performance Tip 5.5

The `break` and `continue` statements, when used properly, perform faster than do the corresponding structured techniques.

Software Engineering Observation 5.2

There is a tension between achieving quality software engineering and achieving the best-performing software. Often, one of these goals is achieved at the expense of the other. For all but the most performance-intensive situations, apply the following rule of thumb: First, make your code simple and correct; then make it fast and small, but only if necessary.

5.8 Logical Operators

- Logical operators
  - Allows for more complex conditions
    - Combines simple conditions into complex conditions
- C++ logical operators
  - && (logical AND)
  - || (logical OR)
  - ! (logical NOT)

5.8 Logical Operators (Cont.)

- **Logical AND (&&) Operator**
  - **Consider the following `if` statement**

```
if ( gender == 1 && age >= 65 )
   seniorFemales++;
```

  - **Combined condition is `true`**
    - **If and only if both simple conditions are `true`**
  - **Combined condition is `false`**
    - **If either or both of the simple conditions are `false`**

Common Programming Error 5.13

Although `3 < x < 7` is a mathematically correct condition, it does not evaluate as you might expect in C++. Use `( 3 < x && x < 7 )` to get the proper evaluation in C++.

expression1	expression2	expression1 && expression2
`false`	`false`	`false`
`false`	`true`	`false`
`true`	`false`	`false`
`true`	`true`	`true`

Fig. 5.15 | && (logical AND) operator truth table.

5.8 Logical Operators (Cont.)

- Logical OR (`||`) Operator
  - Consider the following `if` statement

```
if ( ( semesterAverage >= 90 ) || ( finalExam >= 90 )
   cout << “Student grade is A” << endl;
```

  - Combined condition is `true`
    - If either or both of the simple conditions are `true`
  - Combined condition is `false`
    - If both of the simple conditions are `false`

expression1	expression2	expression1 || expression2
`false`	`false`	`false`
`false`	`true`	`true`
`true`	`false`	`true`
`true`	`true`	`true`

Fig. 5.16 | || (logical OR) operator truth table.

5.8 Logical Operators (Cont.)

- **Short-Circuit Evaluation of Complex Conditions**
  - **Parts of an expression containing && or || operators are evaluated only until it is known whether the condition is true or false**
  - **Example**
    - `( gender == 1 ) && ( age >= 65 )`
      - **Stops immediately if `gender` is not equal to `1`**
        - **Since the left-side is `false`, the entire expression must be `false`**

Performance Tip 5.6

In expressions using operator `&&`, if the separate conditions are independent of one another, make the condition most likely to be `false` the leftmost condition. In expressions using operator `||`, make the condition most likely to be `true` the leftmost condition. This use of short-circuit evaluation can reduce a program's execution time.

5.8 Logical Operators (Cont.)

- **Logical Negation (!) Operator**
  - **Unary operator**
  - **Returns `true` when its operand is `false`, and vice versa**
  - **Example**
    - ```
      if ( !( grade == sentinelValue ) )
         cout << "The next grade is " << grade << endl;
      ```
      is equivalent to:
      ```
      if ( grade != sentinelValue )
         cout << "The next grade is " << grade << endl;
      ```
- **Stream manipulator `boolalpha`**
  - **Display `bool` expressions in words, "true" or "false"**
      ```

Expression	!expression
`false`	`true`
`true`	`false`

Fig. 5.17 | ! (logical negation) operator truth table.

```
// Fig. 5.18: fig05_18.cpp
// Logical operators.
#include <iostream>
using std::cout;
using std::endl;
using std::boolalpha; // causes bool values to print as "true" or "false"

int main()
{
   // create truth table for && (logical AND) operator
   cout << boolalpha << "Logical AND (&&)"
      << "\nfalse && false: " << ( false && false )
      << "\nfalse && true: " << ( false && true )
      << "\ntrue && false: " << ( true && false )
      << "\ntrue && true: " << ( true && true ) << "\n\n";

   // create truth table for || (logical OR) operator
   cout << "Logical OR (||)"
      << "\nfalse || false: " << ( false || false )
      << "\nfalse || true: " << ( false || true )
      << "\ntrue || false: " << ( true || false )
      << "\ntrue || true: " << ( true || true ) << "\n\n";

   // create truth table for ! (logical negation) operator
   cout << "Logical NOT (!)"
      << "\n!false: " << ( !false )
      << "\n!true: " << ( !true ) << endl;
   return 0; // indicate successful termination
} // end main
```

Stream manipulator **boolalpha** causes **bool** values to display as the words “true” or “false”

Use **boolalpha** stream manipulator in **cout**

Output logical AND truth table

Output logical OR truth table

Output logical NOT truth table

fig05_18.cpp

(1 of 2)

```
Logical AND (&&)
false && false: false
false && true: false
true && false: false
true && true: true

Logical OR (||)
false || false: false
false || true: true
true || false: true
true || true: true

Logical NOT (!)
!false: true
!true: false
```

<table>
<tr><th>Operators</th><th>Associativity</th><th>Type</th></tr>
<tr><td>()</td><td>left to right</td><td>parentheses</td></tr>
<tr><td>++ -- static_cast< type >()</td><td>left to right</td><td>unary (postfix)</td></tr>
<tr><td>++ -- + - !</td><td>right to left</td><td>unary (prefix)</td></tr>
<tr><td>* / %</td><td>left to right</td><td>multiplicative</td></tr>
<tr><td>+ -</td><td>left to right</td><td>additive</td></tr>
<tr><td><< >></td><td>left to right</td><td>insertion/extraction</td></tr>
<tr><td>< <= > >=</td><td>left to right</td><td>relational</td></tr>
<tr><td>== !=</td><td>left to right</td><td>equality</td></tr>
<tr><td>&&</td><td>left to right</td><td>logical AND</td></tr>
<tr><td>||</td><td>left to right</td><td>logical OR</td></tr>
<tr><td>?:</td><td>right to left</td><td>conditional</td></tr>
<tr><td>= += -= *= /= %=</td><td>right to left</td><td>assignment</td></tr>
<tr><td>,</td><td>left to right</td><td>comma</td></tr>
</table>

Fig. 5.19 | Operator precedence and associativity.

5.9 Confusing Equality (==) and Assignment (=) Operators

- **Accidentally swapping the operators == (equality) and = (assignment)**
  - **Common error**
    - **Assignment statements produce a value (the value to be assigned)**
    - **Expressions that have a value can be used for decision**
      - **Zero = `false`, nonzero = `true`**
  - **Does not typically cause syntax errors**
    - **Some compilers issue a warning when = is used in a context normally expected for ==**

5.9 Confusing Equality (==) and Assignment (=) Operators (Cont.)

- Example

```
if ( payCode == 4 )
   cout << "You get a bonus!" << endl;
```

  - If `paycode` is 4, bonus is given
- If == was replaced with =

```
if ( payCode = 4 )
  cout << "You get a bonus!" << endl;
```

  - `paycode` is set to 4 (no matter what it was before)
  - Condition is `true` (since 4 is non-zero)
    - Bonus given in every case

Common Programming Error 5.14

Using operator == for assignment and using operator = for equality are logic errors.

Error-Prevention Tip 5.3

Programmers normally write conditions such as `x == 7` with the variable name on the left and the constant on the right. By reversing these so that the constant is on the left and the variable name is on the right, as in `7 == x`, the programmer who accidentally replaces the `==` operator with `=` will be protected by the compiler. The compiler treats this as a compilation error, because you can't change the value of a constant. This will prevent the potential devastation of a runtime logic error.

5.9 Confusing Equality (==) and Assignment (=) Operators (Cont.)

- ***Lvalues***
  - Expressions that can appear on left side of equation
  - Can be changed (i.e., variables)
    - `x = 4;`
- ***Rvalues***
  - Only appear on right side of equation
  - Constants, such as numbers (i.e. cannot write `4 = x;`)
- ***Lvalues*** **can be used as *rvalues*, but not vice versa**

Error-Prevention Tip 5.4

Use your text editor to search for all occurrences of = in your program and check that you have the correct assignment operator or logical operator in each place.

5.10 Structured Programming Summary

- Structured programming
  - Produces programs that are easier to understand, test, debug and modify
- Rules for structured programming
  - Only use single-entry/single-exit control structures
  - Rules (Fig. 5.21)
    - Rule 2 is the stacking rule
    - Rule 3 is the nesting rule

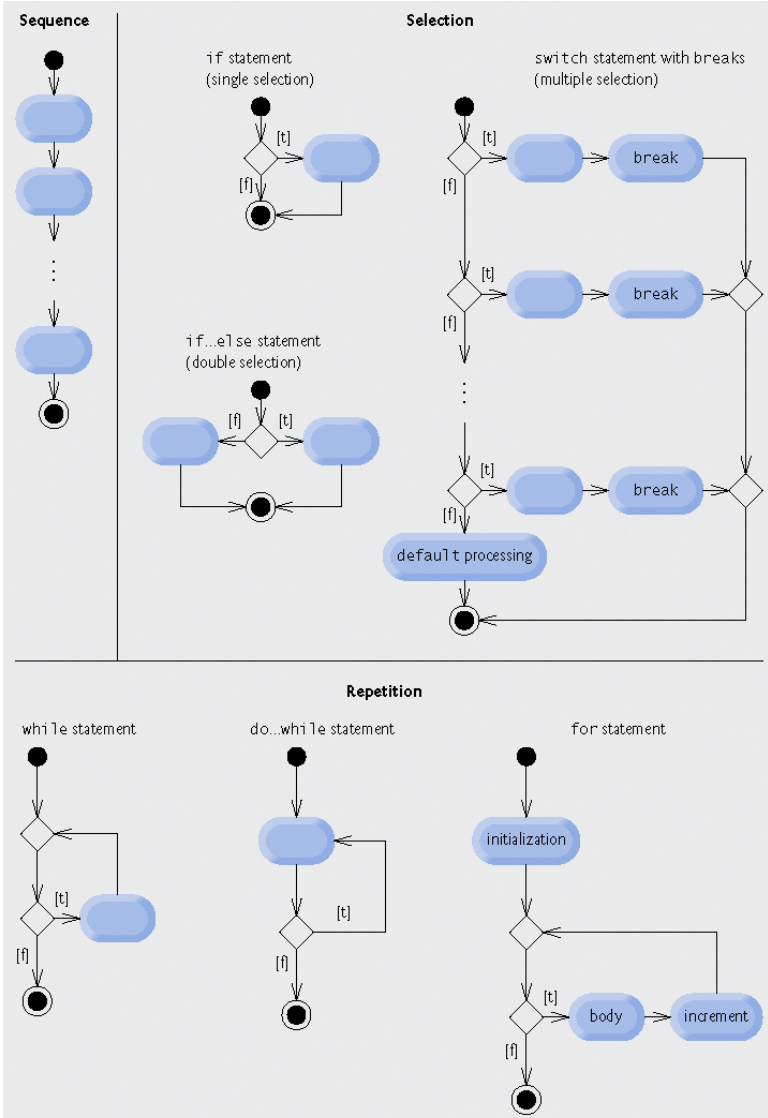

Fig. 5.20 | C++'s single-entry/single-exit sequence, selection and repetition statements.

Rules for Forming Structured Programs	
1)	Begin with the “simplest activity diagram” (Fig. 5.22).
2)	Any action state can be replaced by two action states in sequence.
3)	Any action state can be replaced by any control statement (sequence, `if`, `if...else`, `switch`, `while`, `do...while` or `for`).
4)	Rules 2 and 3 can be applied as often as you like and in any order.

Fig. 5.21 | Rules for forming structured programs.

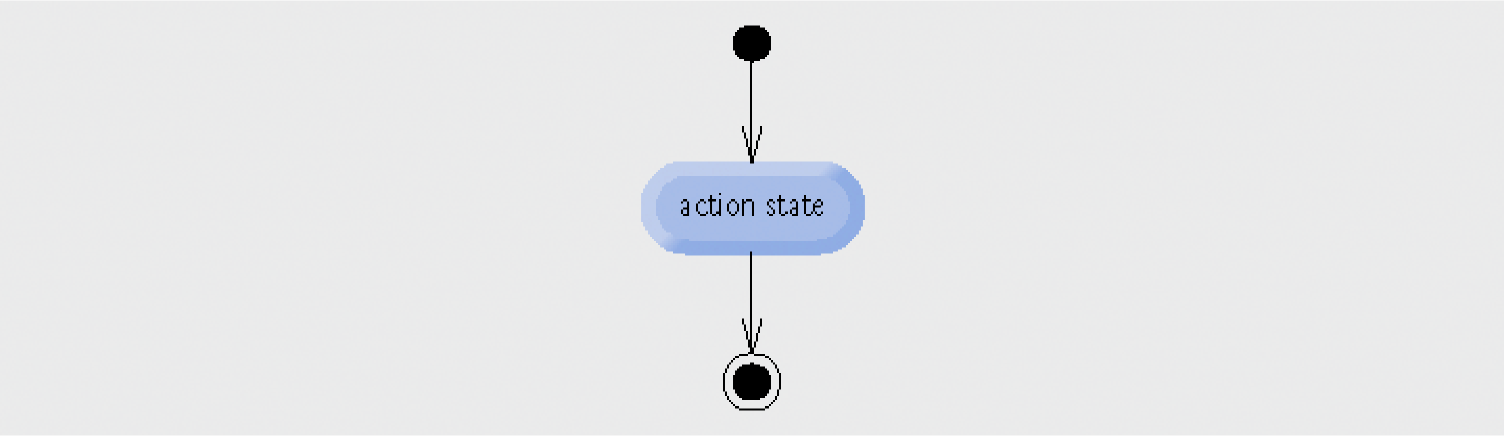

Fig. 5.22 | Simplest activity diagram.

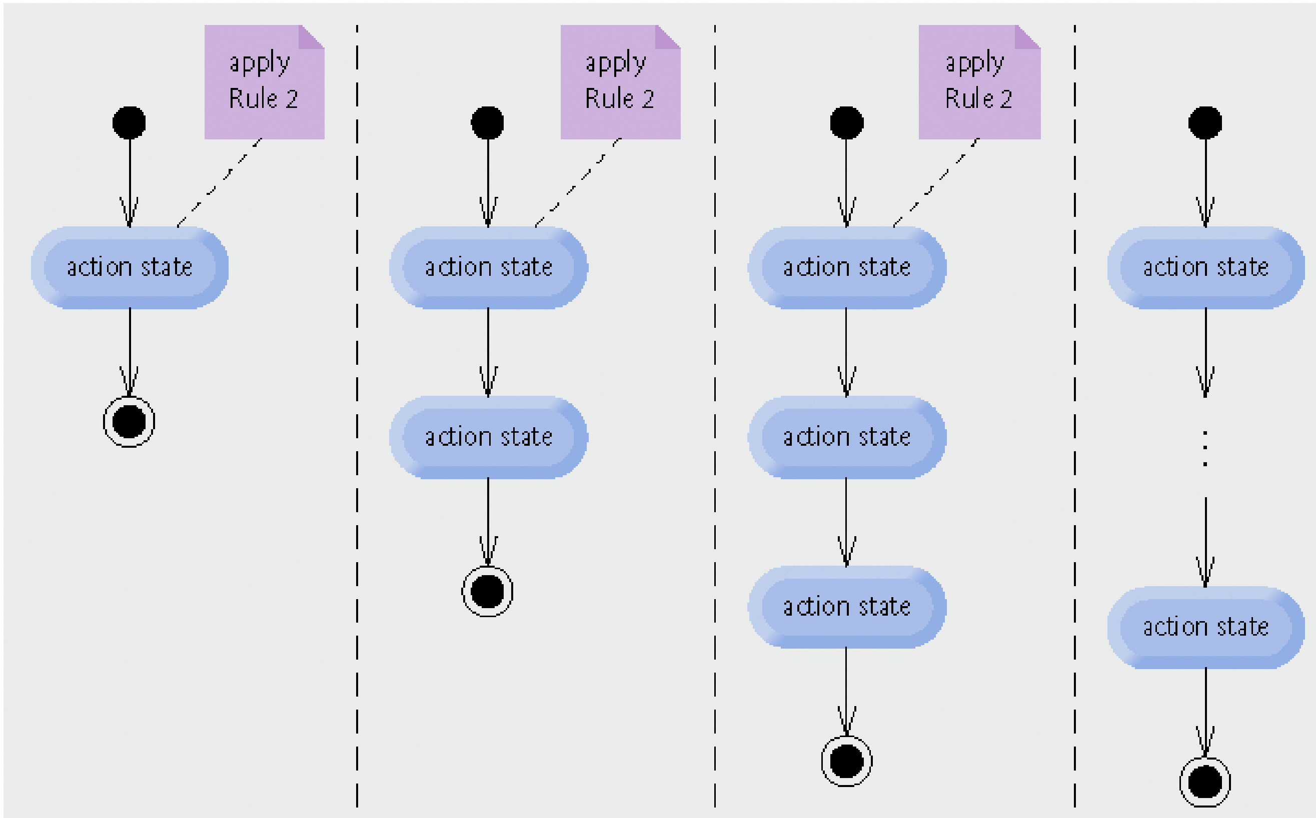

Fig. 5.23 | Repeatedly applying Rule 2 of Fig. 5.21 to the simplest activity diagram.

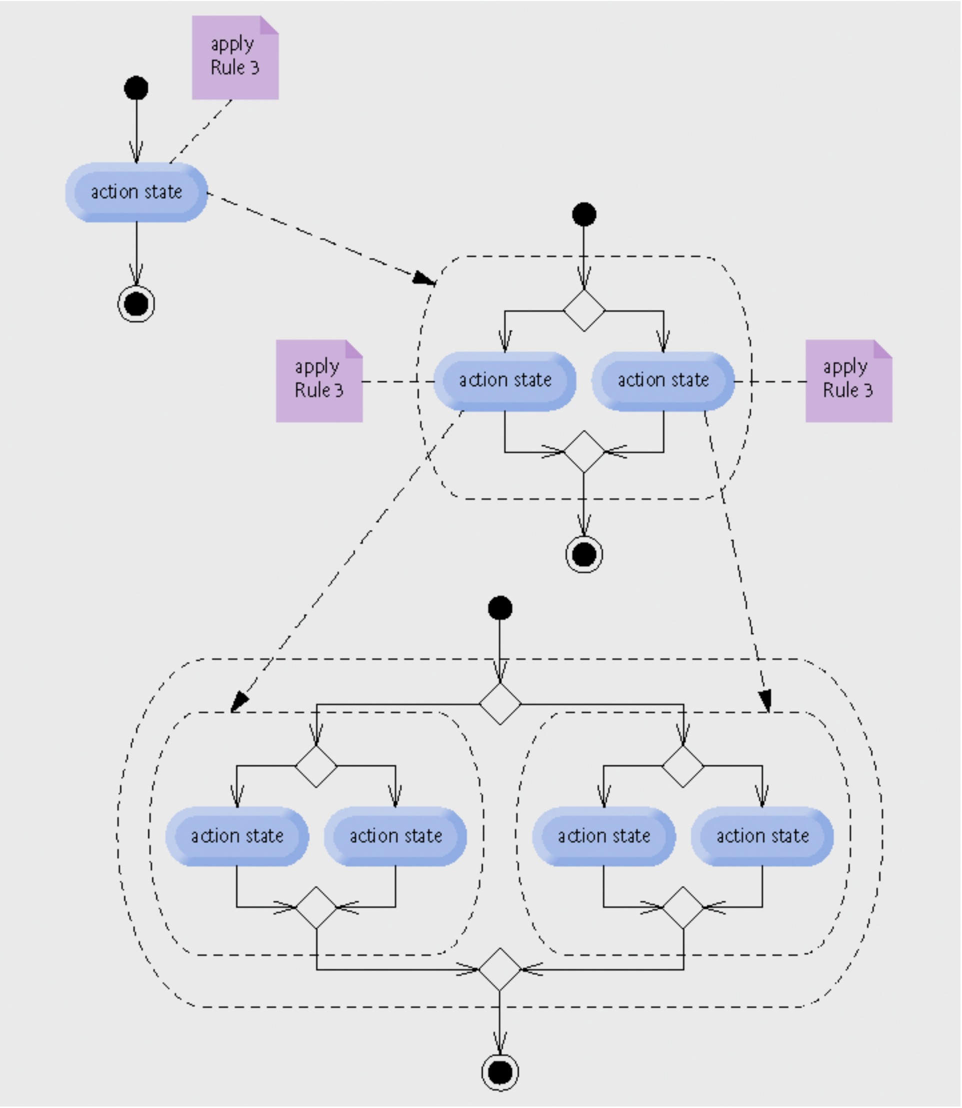

Fig. 5.24 | Applying Rule 3 of Fig. 5.21 to the simplest activity diagram several times.

5.10 Structured Programming Summary (Cont.)

- Sequence structure
  - "built-in" to C++
- Selection structure
  - `if`, `if…else` and `switch`
- Repetition structure
  - `while`, `do…while` and `for`

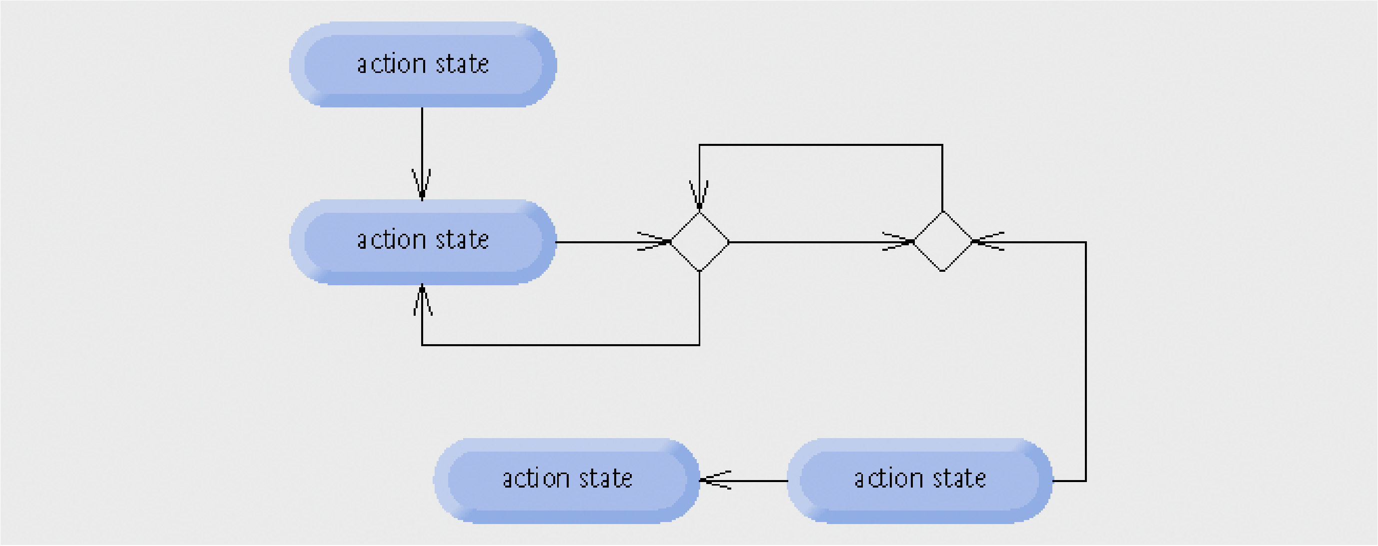

Fig. 5.25 | Activity diagram with illegal syntax.

5.11 (Optional) Software Engineering Case Study: Identifying Object's State and Activities in the ATM System

- **State Machine Diagrams**
  - **Commonly called state diagrams**
  - **Model several states of an object**
    - **Show under what circumstances the object changes state**
    - **Focus on system behavior**
  - **UML representation**
    - **Initial state**
      - **Solid circle**
    - **State**
      - **Rounded rectangle**
    - **Transitions**
      - **Arrows with stick arrowheads**

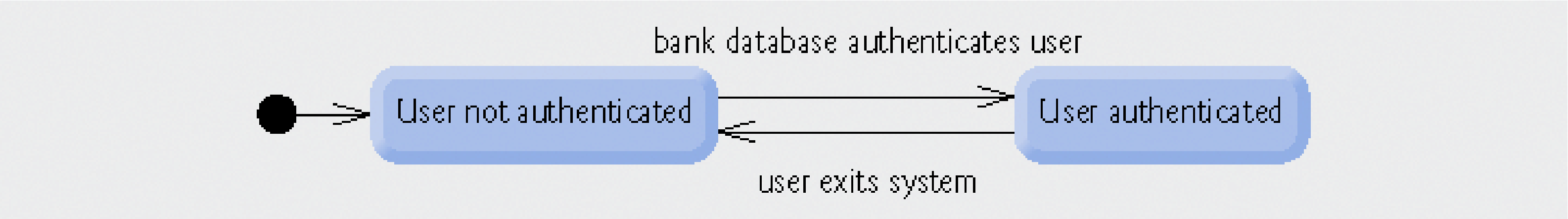

Fig. 5.26 | State diagram for the ATM object.

Software Engineering Observation 5.3

Software designers do not generally create state diagrams showing every possible state and state transition for all attributes—there are simply too many of them. State diagrams typically show only the most important or complex states and state transitions.

5.11 (Optional) Software Engineering Case Study : Identifying Object's State and Activities in the ATM System (Cont.)

- Activity Diagrams
  - Focus on system behavior
  - Model an object's workflow during program execution
    - Actions the object will perform and in what order
  - UML representation
    - Initial state
      - Solid circle
    - Action state
      - Rectangle with outward-curving sides
    - Action order
      - Arrow with a stick arrowhead
    - Final state
      - Solid circle enclosed in an open circle

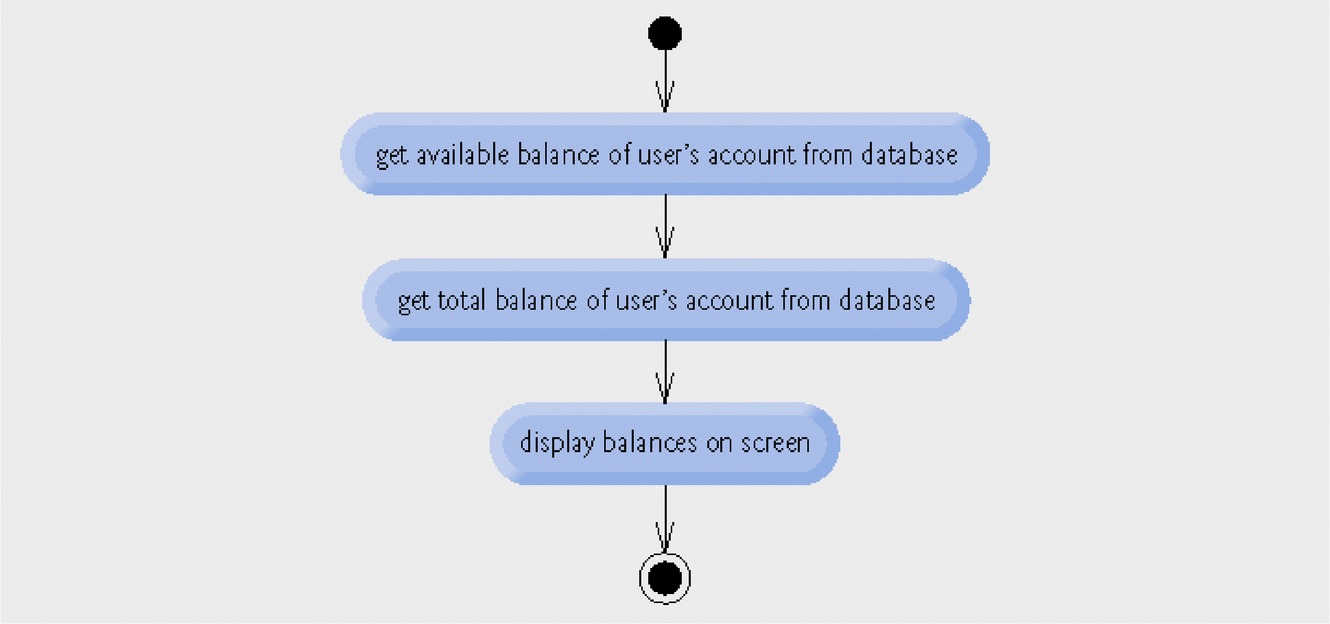

Fig. 5.27 | **Activity diagram for a `BalanceInquiry` transaction.**

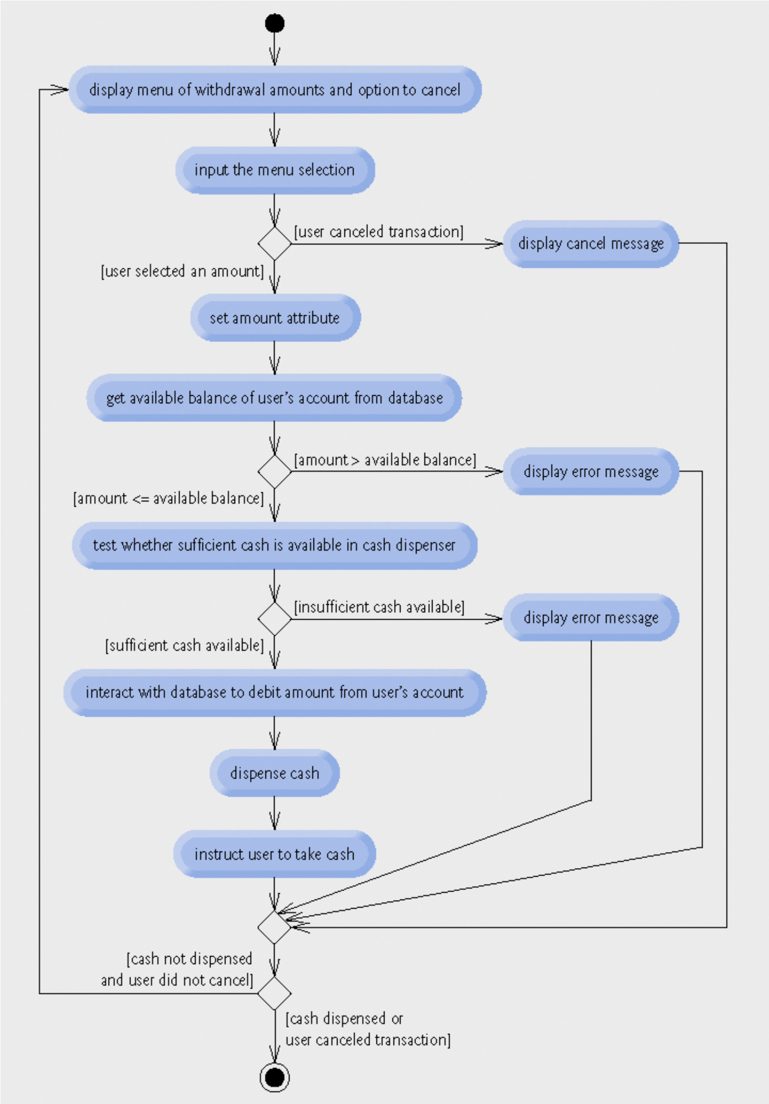

Fig. 5.28 | Activity diagram for a `Withdrawal` transaction.

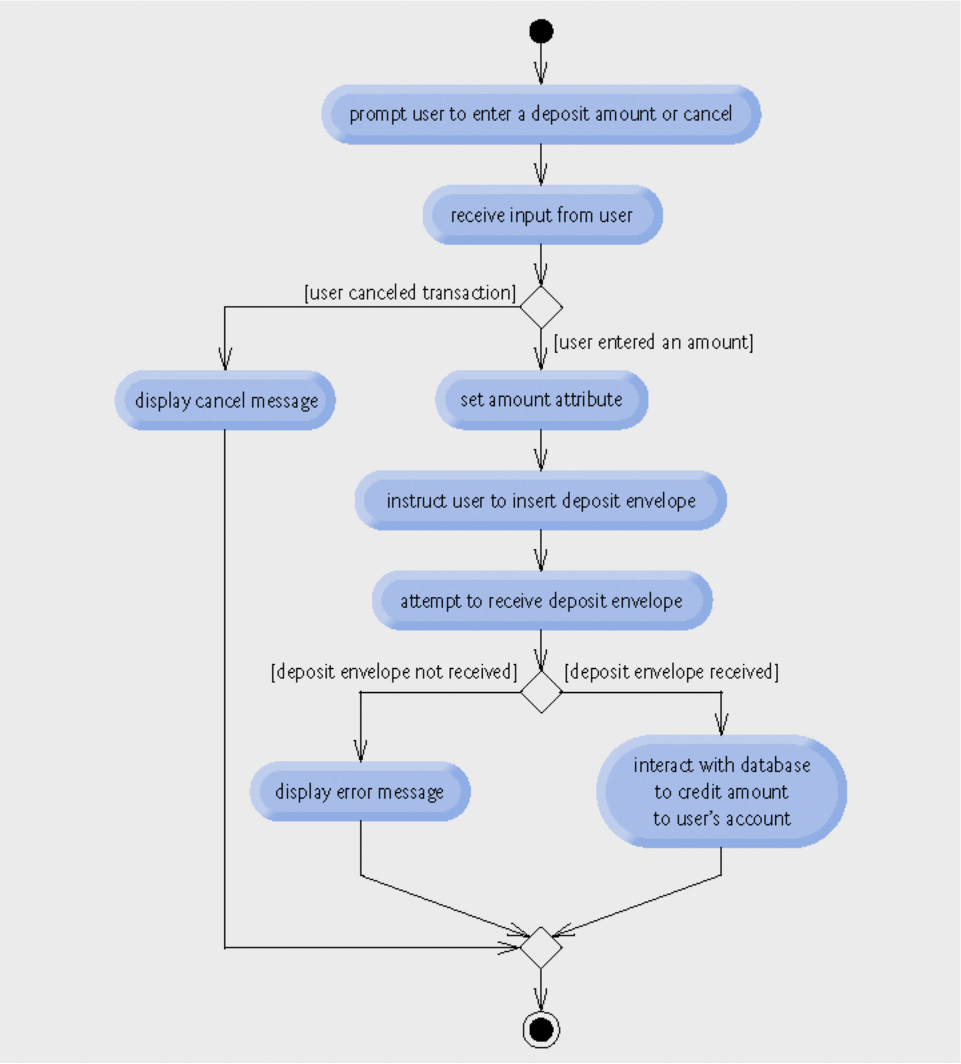

Fig. 5.29 | **Activity diagram for a `Deposit` transaction.**