Programming Design and Implementation

Lecture 3: Debugging and Selection

Updated: 7th March, 2020

Mark Upston
Discipline of Computing
School of Electrical Engineering, Computing and Mathematical Sciences (EECMS)

Outline

Fault Finding

Boolean Operations

IF-THEN-ELSE Statements

CASE Statements

Fault Diagnosis

- The design and implementation of any algorithm should follow these steps:
  - Understand the requirements
  - Design the algorithm
  - Design (and justify) the test data
  - Test the algorithm by hand
  - Implement the algorithm in a programming language
  - Test the program using the **same** test data as with the algorithm
- These steps are sequential, however they will involve revisiting each step many times
- The steps may not be performed by the same person

Fault Diagnosis (2)

- The most common error people make when trying to find and correct an error, is directly trying to discover the error
- In this unit, you must test each class/method in isolation to ensure that it works rigourously
- Systematically test the communication between methods and classes (each connection)
  - See the Addendum slides for testing (`ISE_Testing`) under Resources, for more information.

Fixing Errors

- It is most likely that the first few drafts of an algorithm will contain many errors
- Many programmers refer to these errors as bugs. They are not bugs, they are **mistakes**
- The error could be:
  - In the algorithm (and hence the program)
  - Caused by a difference between the algorithm and the program
  - A programming language specific issue
- The first step is to test all of the assertion statements and narrow down where the errors are
- The next step is to decide which of the above three categories the error belongs to

Fixing the Errors (2)

- Programming Language specific errors are easiest to fix
- Programmer must verify that the correction has achieved the desired result
- Differences between the algorithm and the programming language implementation must be eliminated. This is true even when the difference is not the cause of the error. Consistency is essential
- Errors in the algorithm result in a return to the algorithm design stage. The algorithm must be modified and the modification tested to ensure the problem is resolved.
- Algorithmic errors are the hardest to find and correct

Defensive Algorithm Design

- The art of using as many tools as possible to aid in the elimination of possible causes
  - Consistent formatting of pseudo code and source code
  - Documented contracts between methods (Lecture 4)
  - Documented contracts between objects (Lecture 6)
  - Assertion Statements:
    - A comment placed before a submodule or after a control block which states some condition which is guaranteed to be true after it has been executed
    - Submodule assertion statements are introduced in Lecture 4
    - Control block assertion statements are introduced in this Lecture

Boolean

- Data type which can be **true** or **false**
- Boolean variables are used to reflect an on/off status
- Used with control structures to choose between sections of code (this lecture) or to decide whether or not to repeat a section of code (next lecture)
- There are only two boolean literal constants:
  - **true**
  - **false**
- Examples:

```
boolean isOdd, isPositive;
isOdd = false;
isPositive = true;
```

Relational Operators

Operator	Meaning	Example
==	is equal to	**age** == 50
>	is greater than	**xPos** > **MAX**
<	is less than	**yPos** < **MIN**
>=	is greater than or equal to	**age** >= 75
<=	is less than or equal to	**age** <= 19
!=	is not equal to	**roofColour** != **RED**

- Each of these will evaluate either **true** or **false**
- Each operator has an exact opposite

Boolean Operators

- Three types:
  - `Logical AND` Are both **`true`**
  - `Logical OR` Are either **`true`**
  - `Logical Negation NOT` The opposite of
- Used to combine relational operators to create more complex boolean expressions
- Parenthesis should be used to ensure:
  - The correct order of evaluation will occur
  - The expression is readable
- Syntax for Java is taken from C:
  - Symbol for `Logical AND` is `&&`
  - Symbol for `Logical OR` is `||`
  - Symbol for `Logical Negation` is `!`
- This will evaluate to either, **`true`** or **`false`**

Boolean Operators (2)

Truth Table				
a	b	a&&b	a||b	!b
true	**true**	**true**	**true**	**false**
true	**false**	**false**	**true**	**true**
false	**true**	**false**	**true**	**false**
false	**false**	**false**	**false**	**true**

Boolean Expression Examples

- Given that:

```
int a = 5, b = 10, c = 3;
boolean red = true, brown = false, blue = true;
```

Expression	Result
`red`	`true`
`a > c`	`true`
`(b - a) == c`	`false`
`red || brown`	`true`
`!(red && brown)`	`true`
`(brown && red) || blue`	`true`
`brown && (red || blue)`	`false`
`(a > c) && blue`	`true`
`!(!((b - a) == c))`	`false`
`blue && (b != c)`	`true`

Short Circuit Evaluation

- Java uses Short Circuit Evaluation to minimise the processing required to evaluate boolean expressions
- Only evaluate as much of the expression as required
- `Logical AND`: When the first operand is false, then the entire expression is false and no need to evaluate further
- `Logical OR`: When the first operand is true then the entire expression is true hence no need to evaluate further
- Examples:

```
(x < 10) && (y > 50)
// When (x < 10) is false the expression is false
(life == 42) || (age > 100)
// When (life == 42) is true the expression is true
```

Control Structures

- Control Structures are used to allow portions of an algorithm to be executed under specified conditions
- Two basic types of control structures:
  - Selection: Given one or more possible choices: choose which section (if any) of an algorithm to execute
  - Repetition: Repeat a section of an algorithm provided required conditions are met. Also known as iteration or looping
- Boolean expressions are used to make a choice (selection) or whether to repeat a section of an algorithm (repetition)
- Many programming languages, Java included, allow more than they should. What is important about each control structure is its generic properties. Anything specific to a programming language which is not a reflection of a generic property should be ignored

Selection Control Structures

- Two basic types:
  - The `IF-THEN-ELSE` Statement
    - Provides up to two possible alternatives
  - The `CASE` statement
    - Provides any number of possible alternatives

Properties of IF-THEN-ELSE

- Template:
  - The ELSE part is optional

```
IF boolean_expression THEN
  Statements_for_true
ELSE
  Statements_for_false
END IF
```

- Generic Properties
  - Should be clear that the structure is an **IF-THEN**
  - The boolean expression should be clear
  - What statements are executed if the boolean expression is true should be clear
  - If an **ELSE** clause is included then what statements should be executed if the boolean expression is false should be clear

Properties of IF-THEN-ELSE (2)

- Two basic forms:
- **IF-THEN**
  - Choice is simply perform the action or do not
- **IF-THEN-ELSE**
  - Choice is perform action one or action two
- Note that the syntax for any control structure written in pseudocode does not have to be rigid. It simply has to be clear regarding:
  - What statements are encapsulated by the control structure
  - What is the logical expression controlling the actions taken by the control structure

IF-THEN: Pseudo Code

```
IF booleanExpression THEN
    statement_1
    statement_2
    ...
    statement_n
END IF
// Algorithm Continues
```

```
result = x - y
IF (result < 0.0) THEN
    result = -result
END IF
// Assertion: result will
// be positive
SQUAREROOT result
```

Alternative:

```
IF result < 0 THEN
    negate result
END IF
SQUAREROOT result
```

IF-THEN-ELSE: Pseudo Code

```
IF booleanExpression THEN
    statement_1
    ...
    statement_n
ELSE
    statement_n+1
    ...
    statement_n+m
END IF
// Algorithm Continues
```

```
IF inputPasswd = usrPasswd THEN
    OUTPUT "Access Granted"
    // Give access somehow
ELSE
    OUTPUT "Invalid Entry"
END IF

// Assertion: Access will be
//   granted if the correct
//   password entered
```

IF-THEN: Java Implementation

- No **THEN** required
- Blocks are used to denote the **IF** and **ELSE** parts
- Syntax:

```
if(booleanExpression)
{
    statement_1;
    statement_2;
    ...
    statement_n;
}
// Algorithm Continues
```

```
result = x - y;
if(result < 0.0)
{
    result = -result;
}
// Assertion: result will
// be positive
Math.sqrt(result);
```

IF-THEN: Java Implementation

```
if(booleanExpression)
{
    statement_1;
    ...
    statement_n;
}
else
{
    statement_n+1;
    ...
    statement_n+m;
}
// Algorithm Continues
```

```
if(x - y > 0.0)
    result = x - y;
else
    result = -(x - y);
System.out.println(result);
// Assertion: result will
// be positive
```

- **IF** or **ELSE** parts which consist of one statement do not need to be encapsulated in a block (i.e., `{ ... }`)
- But in PDI you **should** anyway

Nesting IF-THEN-ELSE

- If statements can be nested inside the other
  - So can the other control structures
  - Any control structure can be nested inside any other control structure
- Example:

```
if(x > 0)
    posTally++;
else
    if(x < 0)
        negTally++;
    else
        zeroTally++;
```

- An else is always matched to the nearest unmatched if
  - Which is why you should **always** use blocks ({ ... })

Formatting of code

- The placement of open and close parenthesis should be consistent

Good

```
if((age > 10) && (age < 10))
{
    teenCount++;
    if(offences > 0)
    {
        crimCount++;
    }
    else
    {
        nonCrimCount++;
        pastOffender = false;
    }
}
```

Poor (Unless you are writing a book)

```
if((age > 10) && (age < 18)) {
    teenCount++;
    if(offences > 0)
        crimCount++;
    else {
        nonCrimCount++;
        pastOffender = false;
    }
}
```

Multiple alternatives

- An appropriate format for a multiple alternative decisions where all the decisions related to the same data would be:

```
IF x is positive THEN
    INCREMENT posTally
ELSE IF x is negative THEN
    INCREMENT negTally
ELSE
    INCREMENT zeroTally
ENDIF
```

```
if(x > 0.0)
    posTally++;
else if(x < 0.0)
    negTally++;
else
    zeroTally++;
```

Note: Braces ({ ... }) have been omitted to conserve space

Multiple alternatives (2)

- The previous format is not suitable for multiple alternate decisions where all the decisions relate to different data

```
IF age < 18 THEN
    INCREMENT childTally
ELSE
    IF criminal THEN
        INCREMENT crimTally
    ELSE
        INCREMENT nonCrimTally
    ENDIF
ENDIF
```

```
if(age < 18)
    childTally++;
else
{
    if(criminal)
        crimTally++;
    else
        nonCrimTally++;
}
```

Note: Some braces ({ ... }) have been omitted to conserve space

Ordering of Boolean Expressions

- Need to be careful about the order in which questions are asked
- Consider the example below

```
IF mark >= 80 THEN
    grade = 'H'
ELSE IF mark >= 70 THEN
    grade = 'D'
ELSE IF mark >= 60 THEN
    grade = 'C'
ELSE IF mark >= 50 THEN
    grade = 'P'
ELSE
    grade = 'F'
ENDIF
```

```
IF mark >= 50 THEN
    grade = 'P'
ELSE IF mark >= 60 THEN
    grade = 'C'
ELSE IF mark >= 70 THEN
    grade = 'D'
ELSE IF mark >= 80 THEN
    grade = 'H'
ELSE
    grade = 'F'
ENDIF
```

- Are both of these alternatives valid?
- We need to test these algorithms

Efficiency Considerations

- Order questions from most likely to least likely where possible
- In the previous example, suppose the distribution of student marks is:

Grade	% of Students
High Distinction	5
Distinction	10
Credit	30
Pass	35
Fail	20

- What impact does this have on the ordering of your choices?

Efficiency Considerations (2)

```
IF mark < 50 THEN
    grade = 'F'
ELSE IF mark < 60 THEN
    grade = 'P'
ELSE IF mark < 70 THEN
    grade = 'C'
ELSE IF mark < 80 THEN
    grade = 'D'
ELSE
    grade = 'H'
END IF
```

- Time for more testing

Sequential vs Nested IF's

- Why is the code segment below inefficient?

```
IF x IS POSITIVE THEN
    INCREMENT posTally
END IF
IF x IS NEGATIVE THEN
    INCREMENT negTally
END IF
IF x IS ZERO THEN
    INCREMENT zeroTally
END IF
```

```
if(x > 0)
    posTally++;

if(x < 0)
    negTally++;

if(x == 0)
    zeroTally++;
```

- Novice programmers are often tempted to employ sequential IF's rather than nesting the statements

Beware of Impossible Conditions

- Consider the example below:

```
if((y / x > 10) && (x != 0)) { ... }
```

- What will happen if **x** is equal to zero?
- Taking advantage of short circuit evaluation could lead to:

```
if((x != 0) && (y / x > 10)) { ... }
```

- This can be dangerous, a better alternative would be:

```
if(x != 0)
{
    if(y / x > 10)
    {
        ...
    }
}
```

Real Numbers and Equality

- Comparing real numbers in boolean expressions is no different to comparing other primitive data types, except when using the equality operator
- The potential for error in the lower order decimal places means that any two values may be equal but only within a specified tolerance

  $\sqrt{3} = 1.732050$ and $\sqrt{3} \times \sqrt{3} = 2.999997$
- Deciding what this tolerance should be is an important decision
  - Too large: Just about all numbers are considered equal
  - Too small: Just about all numbers are considered **not** equal
- Two cases which can occur:
  - Testing the equality of whole numbers
  - Testing equality to n decimal places

Comparing Whole Numbers

- Two alternatives:
  - Type convert (truncation will occur)

```
if((int)amount == value)
{
    ...
}
```

  - Use the **round()** method in the **Math** class

```
if(Math.round(amount) == value)
{
    ...
}
```

Comparing Reals within a Tolerance

- The tolerance must be valid for the application
- Two decimal places would be suitable for money but not for inspecting turbine blades
  - **IF amount IS WITHIN** 0.01 **OF** 10.53 **THEN**

```
if(Math.abs(amount - 10.53) < 0.01)
{
    ...
}
```

  - **IF THE DIFFERENCE BETWEEN observedX AND trueX IS LESS THAN blade TOLERANCE THEN**

```
if(Math.abs(trueX - observedX) < blade.TOLERANCE)
{
    ...
}
```

What not to do

- You do **not** need to test if a Boolean is equal to **true**/**false**, they **are** **true**/**false**!

Bad		Good
`if(x > 3 == true)`	=>	`if(x > 3)`
`if(y < 3 == false)`	=>	`if(!(y < 3))`
`if(((x - 3) == 0) == true)`	=>	`if(x - 3 == 0)`
`if(y < 3 != true)`	=>	`if(!(y < 3))`

CASE: Pseudo Code

- Used to provide a number of alternatives where the choice is based upon the result of a single expression

```
Case expression
    case 1    Action1
    case 2    Action2
    ...       ...
    case n    ActionN
    else      ActionN+1
END Case
```

- Generic properties of a **CASE** statement:
  - The expression must involve a discrete data type
    - **int** or **char** (i.e., non-real)
  - Each case must be a list of one or more constant values
  - The only possible comparison is equality
  - Which statements are executed for each set of constant values should be clear

CASE: Java (switch)

- In Java a **case** statement is implemented as a **switch** statement

```
switch(expression)
{
    case const1:
        Statement_Set_1;
    break;
    case const2:
        Statement_Set_2;
    break;
    ...
    case constN:
        Statement_Set_N;
    break;
    default:
        Statement_Set_N+1;
}
```

CASE: Java (switch) (2)

- When the expression matches a case, the statements in the case are executed until:
  - A **break** statement is encountered; or
  - The end of the switch statement is encountered
- The **default** clause is optional
  - If the **default** clause is supplied then it is executed if the **switch** expression does not match any of the case constants
  - If the **default** expression is not supplied and the **switch** expression does not match any of the case constants then the **switch** statement is exited (nothing happens)

Example

```
switch(status)
{
    case 'G':
        student.setStatus("Good Standing");
    break;
    case 'C':
        student.setStatus("Conditional");
    break;
    case 'T':
        student.setStatus("Terminated");
    break;
    default:
        student.setStatus("No Status Recorded");
    break;  // Optional if default is supplied
}
```

Multiple CASE for Same Action

- If the same action is required for more than one case then list all of the cases followed by the required action

```
switch(month)
{
    case 1: case 3: case 5: case 7: case 8: case 10: case 12:
        daysInMonth = 31;
    break;
    case 4: case 6: case 9: case 11:
        daysInMonth = 30;
    break;
    case 2:
        daysInMonth = daysInFeb(year);
    break;  // Not optional as default is not supplied
}
// Assertion: Days in Month will contain
// 31: Jan, Mar, May, Jul, Aug, Oct and Dec
// 30: Apr, Jun, Sep and Nov
// 28/29: Feb
```

Common Mistakes with switch

- Care must be taken to ensure that the **break** statements are placed where they are required
- The compiler cannot check this for you because the **break** statement is not required to follow each case

```
switch(choice)
{
    case '+':
        result = addition(numOne, numTwo);
    case '-':
        result = subtraction(numOne, numTwo);
}
```

- What is the error in the above statement?

IF-THEN-ELSE vs CASE

- Every **switch** can be converted to an **if-else**
- Not every **if-else** can be converted to a **switch**
- Consider the 'Mark/Grade' example (on slides 26/28):
  - Mark is a discreet datatype, so it can be used in a **switch**
  - But there are too many possibilities
  - Need to use a "trick"

Mark/Grade Example

```
newMark = mark DIV 10
CASE newMark

    8: 9: 10:
        grade := 'H'

    7:
        grade := 'D'

    6:
        grade := 'C'

    5:
        grade := 'P'

    DEFAULT:
        grade := 'F'
END CASE
```

```
newMark = mark / 10;
switch(newMark)
{
    case 8: case 9: case 10:
        grade = 'H';
    break;
    case 7:
        grade = 'D';
    break;
    case 6:
        grade = 'C';
    break;
    case 5:
        grade = 'P';
    break;
    default:
        grade = 'F';
}
```

"Zero" Marks

- A student who does any of the following in a submitted, assessable answer will receive heavy penalties, up to and including **zero** marks for that question:
  - Uses **`continue`**
  - Uses **`break`** in any other place than a **`switch`** statement
  - Uses **`goto`** (including **`Exception`**'s for program flow)
  - Has more than one **`return`** statement in a method
  - Has a **`return`** statement in a method anywhere but the last statement of the method
  - Uses **`System.exit()`** anywhere but the last statement of the **`main()`** method
  - Uses global variables for anything other than class fields
  - Uses a ternary operator
- Note: similar efforts in pseudo code will also receive zero marks

Next Week

- The next Lecture will cover:
  - Repetition
    - **`while`**
    - **`do-while`**
    - **`for`**