Guide to Good Pseudocode

Attributes of Good Pseudocode

1. Completeness as a solution
2. General concepts and conventions
3. Clarity, explicitness, and good formatting
4. Concise expression
5. Parsimony
6. Good organisation

1. Completeness as a solution

Pseudocode must solve the problem at hand using well-defined operations. The programmer who translates pseudocode should be able to do so almost mechanically. There should be no need to develop algorithms or solve problems at the programming stage.

Software development is a multi-stage process, and most of the hard work should take place during design, rather than construction. There's an analogy here to writing, say, an essay. The hard part is coming up with your ideas and arguments; getting the grammar and spelling right should be the easy part. In software development, the design of algorithms is the hard part, and the translation to a programming language implementation should be easy.

Bad
Calculate the tax

Good
```
tax = 0
if income > TAX_FREE_THRESHOLD
        tax = income × TAX_RATE
```

2. General concepts and conventions

Pseudocode should not be specific to a particular programming language; it should use general concepts and conventions.

Bad
```
import random module
…
temperature = random.randint(LOW, HIGH)
```

Good
```
temperature = random integer between LOW and HIGH exclusive
```

Why is "import random module" bad? Not every language has a random module, and even if they all did, importing that module is not an essential part of the algorithm. You're not telling the programmer how to write a program, you're telling her how to solve a problem. You can assume that the programmer who implements your pseudocode has enough intelligence and knowledge to use the tools available in her programming language.

3. Clarity, explicitness, and good formatting

Pseudocode needs to be well-indented, well-spaced and be consistent in style, so that it doesn't confuse or annoy the programmer who reads it. Variable names must be consistent and clear, and be used explicitly.

Bad

```
if a    >  B                          poorly spaced
   display message                    unclear (what message?)
    get x                             poor variable name
          if what we got is "y"       not explicit
              vent the gas            not explicit
```

Good

```
if pressure > CRITICAL_PRESSURE       well-spaced and named
        display critical error message clear
        get ventGasChoice              good name
        if ventGasChoice is "y"        explicit
                ventGas()              explicit
```

Note: variable names don't have to be in camel-case (e.g. "vent gas choice" is okay).

4. Concise expression

Pseudocode should be concise. Write instructions as simply as you can without sacrificing clarity.

Bad

```
get input from the user and store it in a variable called "value"
store the number one in a variable called "value"
make c equal to the sum of a and b
```

Good

```
get value
value = 1
c = a + b
```

5. Parsimony

Pseudocode should solve the problem, but that's it! There should be no superfluous material. It's OK to have comments in pseudocode, but the instructions themselves should not speak about the intention.

Bad

```
if x < LEFT_BOUND or x > RIGHT_BOUND then the train could derail
        haltTrain() to stop the train from derailing
```

Good

```
(check if the train is about to derail and stop it)
if x < LEFT_BOUND or x > RIGHT_BOUND
        haltTrain()
```

There should be no instructions that don't achieve anything.

Bad
Define the constants

"Define the constants" is bad because the programmer knows she has to define variables! Such a statement adds no information. Statements like the above often occur when software developers work backwards from the code when writing their pseudocode.

6. Good organisation

Pseudocode should be organised into appropriate functions, as is the case for code. There should be a one-to-one mapping between functions in pseudocode and functions in code. I.e. if *doSomething* is a function in pseudocode, then it should appear as a function in the code, and vice versa.

The pseudocode within functions should have the five quality attributes already discussed. Each individual function should be specified separately; the pseudocode for a function should not be written where it is called.

Pseudocode Patterns

Comments in pseudocode

If comments are necessary, write them on their own lines, surrounded by parentheses.

- *(check if the train is about to derail and stop it)*

Terminal input: how to get data into a variable

- get name
- get age

Terminal output: how to display a variable or message

- display name
- display venting gas message
  Note: it is not necessary to replicate the interface in pseudocode. The programmer should have access to that part of the planning, and so will know what the "venting gas message" is. Including large quotes makes pseudocode harder to read.

Arithmetic

- gross = hours * rate
- nett = gross – tax
- average = total / count (use floating point arithmetic)

Selection

- if age >= 65
  price = price – seniorDiscount
- if temperature > 60
  display "Too hot!"
  otherwise if temperature < 40
  display "Too cold!"
  otherwise
  display "Just right ☺"
  Note: strings like "Too hot!" are very short, and so they don't detract from the readability of the pseudocode, and hence can be directly included.

Repetition

- repeat while price > 0
  total = total + price
  get price
- for count from 1 to 10
  display count
- for each grade in gradeList
  display grade
- repeat n times
  display horizontal line

Function definitions

- function doSomething(x, y)
  return (x + y) * (y – x)

Function calls

- doSomething(azimuth, altitude)

File input

- open “info.txt” as fileIn for reading
  get name from fileIn
  get age from fileIn
  close fileIn

File output

- open “stats.dat” as fileOut for writing
  for each datum in data
  write datum to fileOut
  write newline to fileOut
  close fileOut

String manipulation

- response = response in uppercase
- words = split sentence on whitespace
- commaPosition = find first comma in sentence

Lists

- priceList = empty list
- add price to priceList
- priceList[0] = newPrice
- if index < length of priceList
  display priceList[index]

Maps (a.k.a. dictionaries or associative arrays)

- phoneNumbers = empty map
- phoneNumbers[name] = number
- for each name in the domain of phoneNumbers
  display name
- for each number in the image of phoneNumbers
  display number

- for each name:number in phoneNumbers
  display name and number

Classes

- class Person
  constructor(name, age, gender)
  instance.setName(name)
  instance.setAge(age)
  instance.setGender(gender)

  basic getters for name, age, gender

  basic setters for name, age

  method setGender(gender)
  if gender in lowercase is not “male” or “female”
  gender = “female”
  instance.gender = gender

- **Note:** getName() etc. are so simple that it’s only necessary to specify that they’ll exist (i.e. ***basic getters for name, age, gender***), rather than writing pseudocode.