Chapter 7

End of Chapter Exercises

1. *Think of a real-world piece of machinery that you use regularly. It could be a VCR, a games console, even a washing machine. Now view the item as if it were a software object: list its methods (the things it can do) and its properties (the information it needs to do its job — some of this might be represented on its display screen/light panel if it has one).*

No solution given -- anything could have been chosen.

2. *Design algorithms for each of the methods for the* `Person` *class.*

The `Person` class outline was like this:

class `Person`

Properties

```
awake: yes, no ;
inBed: yes, no ;
needsShower: yes, no ;
isDressed: yes, no ;
```

Methods

```
WakeUp ;
GoToSleep ;
GetUp ;
GoToBed ;
GetWashed ;
GetDressed ;
GetUndressed ;
```

Method algorithms

GoToSleep

```
1.  awake ← No ;
```

GetUp

```
1.  inBed ← No ;
```

GoToBed

```
1.  inBed ← Yes ;
```

GetWashed
```
1.  needsShower ← No ;
```

GetDressed
```
1.  isDressed ← Yes ;
```

GetUndressed
```
1.  isDressed ← No ;
```

3. *Design algorithms for each of the methods for the* `Alarm` *class.*

The `Alarm` class outline was like this:

class `Alarm`

Properties
```
ringing: yes, no ;
time: 00:00:00 to 23:59:00 ;
alarmTime: 00:00:00 to 23:59:00 ;
alarmIsSet: on, off ;
```

Methods
```
SetTime hh:mm:ss ;
GetTime ;
SetAlarmTime: hh:mm ;
GetAlarmTime ;
SetAlarm ;
UnsetAlarm ;
StartRinging ;
SwitchOff ; (i.e. stop ringing)
```

Method algorithms

SetTime: hh:mm:ss
```
1.  time ← hh:mm:ss ;
```

GetTime
```
1.  Display time ;
```

SetAlarmTime: hh:mm
```
1.  alarmTime ← hh:mm ;
```

GetAlarmTime

```
1.  Display alarmTime ;
```

SetAlarm

```
1.  alarmIsSet ← on ;
```

UnsetAlarm

```
1.  alarmIsSet ← off ;
```

StartRinging

```
1.  ringing ← yes ;
```

SwitchOff

```
1.  ringing ← no ;
```

4. *Assume our* `Person` *class has another method,* `BedStatus`*, that tells us the value of the* `inBed` *property. If we had several instances of the* `Person` *class, what would the algorithm look like that counts up how many of the* `Person` *objects are still in bed?*

```
1.  stillInBed ← 0 ;
2.  WHILE (Person objects to look at)
       2.1  tell Person BedStatus: answer ;
       2.2  IF (answer = 'yes')
                2.2.1  stillInBed ← stillInBed + 1 ;
            ENDIF
       2.3  Move to next Person object ;
    ENDWHILE
4.  Display stillInBed ;
```

5. *We defined the* `Person` *class to contain methods dealing with going to bed as well as getting up in the morning. Extend the controller algorithm in Solution 7.7 to show the* `brian` *object going to bed. Also, assume that the controller is being run on a Friday and that on Saturdays Brian sleeps late which requires his alarm to be reset to the later time of 10.00 a.m.*

Here's the original solution 7.7:

```
1.  brian ← new Person;
2.  briansAlarm ← new Alarm ;
3.  tell briansAlarm SetTime: currentTime ;
```

```
4.  tell briansAlarm SetAlarmTime: "07:00:00" ;
5.  wait until briansAlarm ringing property = "Yes" ;
6.  tell brian WakeUp ;
7.  tell briansAlarm SwitchOff ;
8.  tell brian GetUp ;
9.  tell brian GetWashed ;
10. tell brian GetDressed
```

Here's my modified version to incorporate the above requirements (changes shown in bold):

```
1.  brian ← new Person;
2.  briansAlarm ← new Alarm ;
3.  tell briansAlarm SetTime: currentTime ;
4.  tell briansAlarm SetAlarmTime: "10:00:00" ;
5.  tell brian GoToBed ;
6.  wait until briansAlarm ringing property = "Yes" ;
7.  tell brian WakeUp ;
8.  tell briansAlarm SwitchOff ;
9.  tell brian GetUp ;
10. tell brian GetWashed ;
11. tell brian GetDressed
```

6. *We treated the problem of getting dressed as a single action. If we assume that the task involves putting on underwear, socks, trousers, a shirt, and shoes:*

i) define classes for each of these different clothing types (Underwear, Socks, Trousers, Shirt, Shoes). Think about what properties and methods each clothing class should have.

ii) instantiate the following objects belonging to the different clothing classes: `tanPleats` *(Trousers),* `whiteBoxers` *(Underwear),* `blackAnkles` *(Socks),* `brownBrogues` *(Shoes),* `whiteLongSleeve` *(Shirt).*

iii) Extend your solution to pass messages to each of these clothing objects instructing them to be `PutOn`*.*

i) Clothing classes

class Socks

Properties

```
beingWorn: yes, no ;
dirty: yes, no
```

wholePair: yes no **// one sock may be missing**

Methods
PutOn ;
TakeOff ;
Wash ;
Dispose ; **// if one sock is missing!**

class Underwear

Properties
beingWorn: yes, no ;
dirty: yes, no ;

Methods
PutOn ;
TakeOff ;
Wash ;

class Trousers

Properties
beingWorn: yes, no ;
dirty: yes, no ;
belt ; **// (class)**

Methods
PutOn ;
TakeOff ;
DryClean ;
ThreadBelt ;
RemoveBelt ;

class Shirt

Properties
beingWorn: yes, no ;
dirty: yes, no ;
wrinkled: yes, no ;

Methods
PutOn ;

```
TakeOff ;
Wash ;
Iron ;
```

class Shoes

Properties

```
beingWorn: yes, no ;
dirty: yes, no ;
wholePair: yes, no ;
Laces ; // (class)
```

Methods

```
PutOn ;
TakeOff ;
Polish ;
ThreadLaces ;
RemoveLaces ;
Dispose ;
```

ii) Instantiating of objects

```
tanPleats ← new Trousers ;
whiteBoxers ← new Underwear ;
blackAnkles ← new Socks ;
brownBrogues ← new Shoes ;
whiteLongSleeve ← new Shirt ;
```

iii) Calling methods.

```
tell tanPleats PutOn ;
tell whiteBoxers PutOn ;
tell blackAnkles PutOn ;
tell brownBrogues PutOn ;
tell whiteLongSleeve PutOn ;
```

7. *Suggest some other methods that could sensibly be included in the* `Person` *class and design algorithms for those methods.*

Any sensible method will do. How about `CelebrateSignificantBirthday`?

CelebrateSignificantBirthday

```
1.  IF (today = birth date and birth month)
1.1.  age ← age + 1 ;
    1.2.  IF (age = 18)
          1.2.1. // Celebrate coming of age
    1.3.  ELSE IF (age = 21)
             1.3.1. // Celebrate 21
    1.4.  ELSE IF (age = 30)
             1.4.1. // celebrate 30
      ....
      etc. etc.
      ENDIF
```

Projects

StockSnackz Vending Machine

Look at the vending machine problem through an object-oriented lens. Suppose we decide that there are three classes involved in a vending machine: the Snacks *it dispenses, a* Vendor *mechanism for dispensing the snacks, and the* MoneyHandler *that receives coins, ensures sufficient money has been paid, and gives change. The* MoneyHandler *would also have to tell the* Vendor *mechanism to release a* Snack*. Try defining the methods and properties for each of these three classes.*

Here's a first go. Note, I haven't been exhaustive and there are still a few requirements to be dealt with (such as checking if sufficient money has been paid), but this should be enough to get you on the way to completing it all.

class Snack

Properties

```
price:  {0...99} ;
name:  {25 characters} ;
stockLevel: {0...25} ;
soldCount: {0...?} ;
```

Methods

```
Dispense ;
Restock: amount ;
SetPrice: amount ;
ShowStockLevel ;
ShowSoldCount ;
```

Method algorithms

Dispense
```
1.  stockLevel ← stockLevel – 1 ;
2.  soldCount  ← soldCount + 1 ;
```

Restock: amount
```
1.  stockLevel  ← stockLevel + amount ;
```

SetPrice: amount
```
1.  price  ← amount ;
```

ShowStockLevel
```
1.   ← stockLevel ;
```

ShowSoldCount
```
1.   ← soldCount ;
```

class Vendor

Properties
```
snacksDispensed: {0...?} ;
...
```

Methods
```
Vend: snack ;
...
```

Method algorithms

Vend
```
1.  IF (snackChoice = Button1)
       1.1. tell button1Snack Dispense ;
2.  ELSE IF (snackChoice = Button2)
       2.1. tell Button2Snack Dispense ;
3.  ELSE IF (snackChoice = Button3)
       3.1. tell Button3Snack Dispense ;
4.  ELSE IF (snackChoice = Button4)
       4.1. tell Button4Snack Dispense ;
5.  ELSE IF (snackChoice = Button5)
       2.1. tell Button5Snack Dispense ;
    ENDIF
```

```
6.  snacksDispensed ← snacksDispensed + 1 ;
```

class MoneyHandler

Properties

```
fiftyPenceCoinStock: {0...100} ;
twentyPenceCoinStock: {0...200} ;
tenPenceCoinStock: {0...300} ;
fivePenceCoinStock: {0...300} ;
twoPenceCoinStock: {0...250} ;
onePennyCoinStock: {0...500} ;
floatValue: {0...?} ;
moneyTaken: {0...?} ;
```

Methods

```
Receive50p: numberCoins ;
Receive20p: numberCoins ;
...
Receive1p: numberCoins ;
Dispense50p: numberCoins ;
...
Dispense1p: numberCoins ;
DisplayFloatValue ;
DisplayMoneyTaken ;
GiveChange: amount ;
VendSnack: snackChoice ;
...
```

Method algorithms

Receive50p: numberCoins

```
1.  fiftyPenceCoinStock ← fiftyPenceCoinStock + numberCoins ;
```

…

DisplayFloatValue

```
1.  ← floatValue ;
```

DisplayMoneyTaken

```
1.  ← moneyTaken ;
```

GiveChange: amount

```
1.  Dispense50p: amount ÷ 50 ;
2.  remainder  ← amount MOD 50 ;
3.  Dispense20p: remainder ÷ 20 ;
4.  remainder  ←  remainder MOD 20 ;
5.  Dispense10p: remainder ÷ 10 ;
6.  remainder ← remainder MOD 10 ;
7.  Dispense5p: remainder ÷ 5 ;
8.  remainder ← remainder MOD 5 ;
9.  Dispense2p: remainder ÷ 2 ;
10. Dispense1p: remainder MOD 2 ;
```

Dispense50p: numberCoins

```
1.  FOR counter GOES FROM 1 TO numberCoins
    1.1.  drop 50p coin into chute ;
    1.2.  fiftyPenceCoinStock ← fiftyPenceCoinStock – 1 ;
    ENDFOR
```

...

VendSnack

```
1.  tell Vendor Vend: snackChoice ;
```

Stocksfield Fire Service: Hazchem Signs

No exercise

Puzzle World: Roman Numerals and Chronograms

No exercise.

Pangrams: Holoalphabetic Sentences

No exercise

Online Bookstore: ISBNs

No exercise