Contents

Interfaces

Motivation

Imagine you want to represent a variety of devices, and comes up with the following UML diagram:

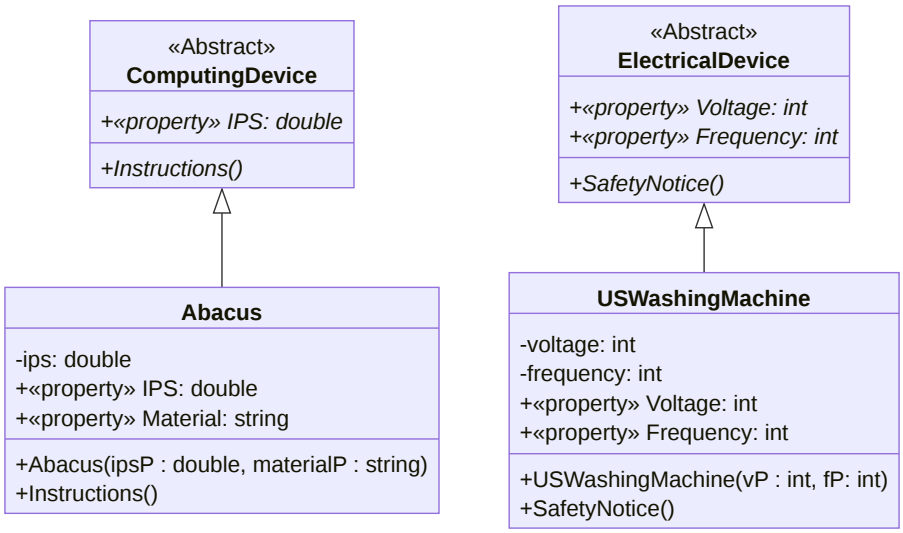

Figure 1: A UML diagram for the ComputingDevice ◁– Abacus class (text version[2])

Note that it is possible to gather that e.g., the `Instructions()` method in the `Abacus` class is overriding the `Instructions()` method in the `ComputingDevice` class because it has the same signature: this can be the case only because it is overriding the inherited abstract method.

Your abstract classes are "completely abstract", in the sense that all of their properties and methods are abstract, but it serves your purpose just well:

- you do not want "ComputingDevices" to exist in isolation, but you want any class representing a computing device such as the Abacus[3], the Pascaline[4] or the Turing Tumble[5] to have an *Instruction per seconds* property, and a method to display instructions.
- similarly, you want any "ElectronicalDevice" to have properties pertaining to their voltage and frequency, as well as a method to display a safety notice.

A class that is "completely abstract" actually forces you to enforce a series of constraints and is a good way of making sure that you are consistent e.g., with the naming of your methods, the accessibility of your properties, or the return type of your methods.

You implement it as follows:

```
abstract class ComputingDevice
{
  public abstract double IPS { get; set; }
  public abstract void Instructions();
}

using System;

class Abacus : ComputingDevice
{
  private double ips;
  public override double IPS
  {
    get { return ips; }
    set
    {
      if (value < 0 || value > 1000)
        throw new ArgumentException(
          "This is not plausible"
        );
      else
        ips = value;
    }
  }
  public string Material { get; set; }

  public Abacus(double ipsP, string materialP)
  {
    IPS = ipsP;
```

[3] https://en.wikipedia.org/wiki/Abacus
[4] https://en.wikipedia.org/wiki/Pascal%27s_calculator
[5] https://en.wikipedia.org/wiki/Turing_Tumble

```
    Material = materialP;
  }

  public override void Instructions()
  {
    Console.WriteLine(
      "Refer to https://www.wikihow.com/Use-an-Abacus"
    );
  }
}

abstract class ElectricalDevice
{
  public abstract int Voltage { get; set; }
  public abstract int Frequency { get; set; }
  public abstract void SafetyNotice();
}

using System;

class USWashingMachine : ElectricalDevice
{
  private int voltage;
  public override int Voltage
  {
    get { return voltage; }
    set
    {
      if (value < 110 || value > 220)
      {
        throw new ArgumentOutOfRangeException();
      }
      else
      {
        voltage = value;
      }
    }
  }
  private int frequency;
  public override int Frequency
  {
    get { return frequency; }
    set
    {
      if (value != 50 && value != 60)
      {
        throw new ArgumentOutOfRangeException();
```

```
      }
      else
        frequency = value;
    }
  }

  public USWashingMachine(int vP, int fP)
  {
    Voltage = vP;
    Frequency = fP;
  }

  public override void SafetyNotice()
  {
    Console.WriteLine(
      "Refer to https://www.energy.gov/sites/"
        + "prod/files/2016/06/f32/"
        + "NFPA_DryerWasherSafetyTips.pdf"
    );
  }
}
```

```
using System;

class Program
{
  static void Main()
  {
    Abacus test0 = new Abacus(1.5, "Wood");
    test0.Instructions();
    USWashingMachine test1 = new USWashingMachine(120,
↪   50);
    test1.SafetyNotice();
  }
}
```

(Download this code)[6]

Then, you would like to add a "Computer" class, but face an issue: classes can inherit only from one class directly, but of course a computer is *both* an electrical device **and** a computing device. A solution is to switch to *interfaces*.

[6] https:/princomp.github.io/code/projects/Devices.zip

Explanations

Interfaces are completely abstract classes: they do not implement anything, they simply force classes inheriting from them (we actually say *that realizes them*) to implement certain features.

In Diagram

Interfaces are prefixed by the «`Interface`» mention, and have all their properties and methods marked as abstract (so, in *italics*). A class can "inherits" from multiple interface (we say that it **realizes** multiple interfaces), and this is marked with an arrow with an open triangle end and a dashed line[7].

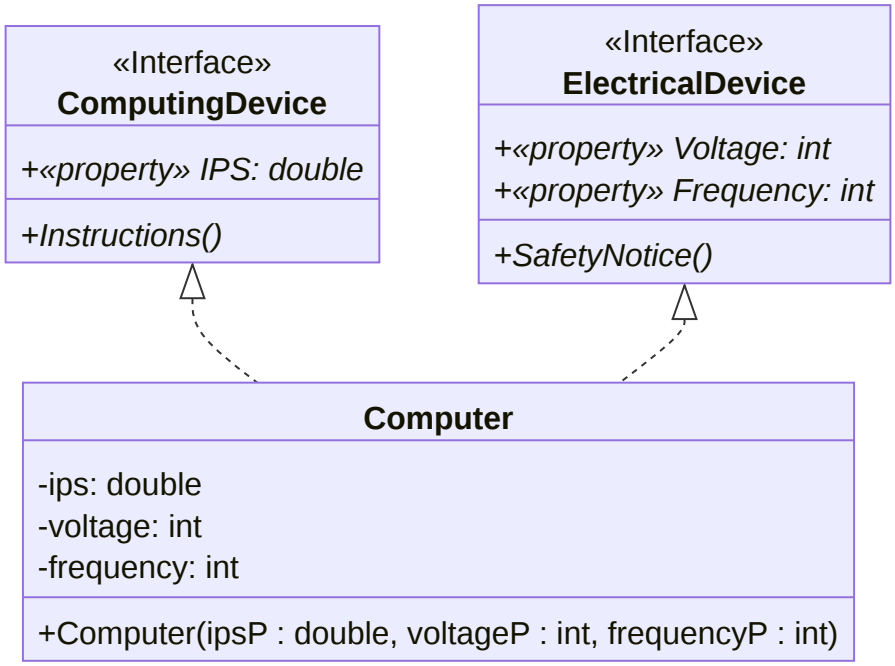

Figure 2: A UML diagram for the ComputingDevice ◁--- Computer class (text version[9])

An Implementation

Implementing such interfaces and their realization could be done as follows:

[7] Note that, this time, since our code below does **not** override the methods and properties, there really is no need to repeat them the derived classes.

```
interface ComputingDevice
{
  double IPS { get; set; }
  void Instructions();
}

interface ElectricalDevice
{
  int Voltage { get; set; }
  int Frequency { get; set; }
  void SafetyNotice();
}

using System;
class Computer: ElectricalDevice, ComputingDevice
{
    private double ips;
    public double IPS
    {
        get { return ips; }
        set
        {
            if (value < 0)
                throw new ArgumentException(
                  "This is not possible."
                );
            else
                ips = value;
        }

    }
    private int voltage;
    public int Voltage
    {
        get { return voltage; }
        set
        {
            if (value < 110 || value > 220)
            {
                throw new ArgumentOutOfRangeException();
            }
            else
            {
                voltage = value;
            }
        }
    }
```

```
    private int frequency;
    public int Frequency
    {
        get { return frequency; }
        set
        {
            if (value != 50 && value != 60)
            {
                throw new ArgumentOutOfRangeException();
            }
            else
                frequency = value;
        }
    }
    public Computer(double ipsP, int voltageP, int
     ↪  frequencyP)
    {
        IPS = ipsP;
        Voltage = voltageP;
        Frequency = frequencyP;
    }
    public void Instructions()
    {
        Console.WriteLine(
          "Refer to your operating system manual."
        );
    }
    public void SafetyNotice()
    {
        Console.WriteLine("Refer to your manufacturer
 ↪  website.");
    }
}
using System;

class Program
{
    static void Main()
    {
        Computer test0 = new Computer(100000, 120, 50);
        test0.SafetyNotice();
        test0.Instructions();
    }
}
```

(Download this code)[10]

Note that

- in the `ComputingDevice` and `ElectricalDevice`,
  - **`abstract class`** has been replaced by **`interface`**,
  - there is no need for the **`abstract`** keyword (all is abstract already!),
  - there is no need for the **`public`** keyword (everything *has* to be public),
- in the `Computer` realization,
  - the class *realizes* two interfaces, simply separated by a comma:
    `Computer: ElectricalDevice, ComputingDevice`
  - there is no need for the **`override`** keyword,

A More Complicated Example

in this archive[13]

[10] https:/princomp.github.io/code/projects/DevicesInterfaces.zip
[13] https:/princomp.github.io/code/projects/Shape.zip

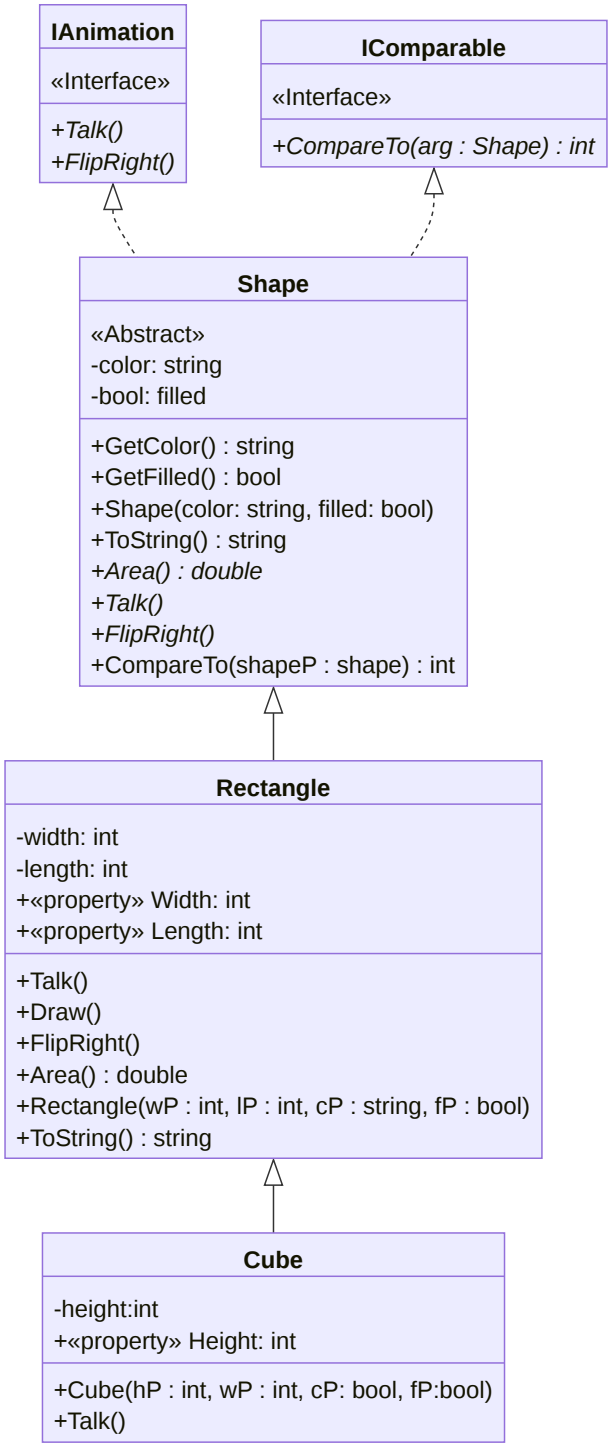

Figure 3: A UML diagram for the IAnimation ◁--- Shape class (text version[12])