Technology-Friendly Programming

Rob Campbell
Whitireia Polytechnic
Porirua City, New Zealand
r.campbell@whitireia.ac.nz

ABSTRACT

For programmers, the only constant is technology-driven change. For this reason, it is appropriate that the coding we teach and produce is technology-friendly. That is, its style can be adapted for use in a scientific or engineering environment. In this paper I discuss the appropriateness of some programming techniques in scientific environments, and suggest promoting three specific skills: economy of expression, program efficiency, and a greater emphasis on procedural approaches.

It is not argued that object orientation should be diluted or minimised within our courses.

KEYWORDS

Scientific programming, procedural, object orientated, variable name, economy.

1. INTRODUCTION

Imagine: it's 1976, and you have to devise a meaningful course in computing for your students. You want to equip them with skills that will still have relevance in twenty-five years time. Computing machinery will probably be very different by 2001 (which will be the winner: paper tape or punched cards?) So will the environment in which your students are employed.

We all know what happened: the employment scene was transformed beyond recognition. There was one constant – the change itself, and the technology powering it. Twenty-five years ahead, we can expect continuing advances, and therefore employment, in science and engineering. That's more than can be said for any other area. If you don't believe me, ask your local unemployed bank manager. Back in 1976, it all looked so secure.

It follows that our programming techniques should be relevant to the arenas of science and engineering. Yet the requirements in these areas often differ from those of commercial programming. For the purpose of this paper, I'll suggest three areas where the needs of the two communities differ: economy of expression, speed of execution, and object orientation.

2. ECONOMY OF EXPRESSION

Teachers and textbooks frequently advocate the use of long, meaningful names, sometimes with Hungarian prefixes thrown in. In scientific programming, this is not always a good idea. Which of these code segments in Figure 1 do you prefer ?

From a coding perspective, the two are identical. You can probably dredge up enough high school science to follow the reasoning behind the second version, but the first…

The formulas in the first case are obscured by the names, whereas in the second they are quite readable. Since in this situation, it's the formulas that are most likely to conceal an error, these should always be presented in the most familiar way using, wherever possible, short variable names.

The aim remains the same: clarity of expression.

3. SPEED OF EXECUTION

Much commercial software is produced rapidly. Unfortunately it is unlikely to run rapidly. Rapid application development tools are appropriate for many environments; time-critical situations such as process control and data acquisition are not among them. Students should be capable of writing lean, efficient code, and should appreciate that elaborate user interfaces may carry unacceptable speed overheads.

4. PROCEDURAL VS OBJECT ORIENTATED CODE

Like any other, scientific and engineering applications benefit from this approach too. But there remains a wide range of situations where object orientation can be positively unhelpful. Algorithms required to calculate orbital ellipticity or stress tensors tend to be 'one off'; there is no need to spawn objects or inherit methods. They are an unnecessary computational overhead, and a procedural approach is recommended. It follows that students should be familiar with procedural programming.

I am not proposing a return to spaghetti code or a rejection of the advances of the last decade. Nonetheless, students can gain the impression that object orientation is the only approach; that it solves all problems and is universally applicable. This is not the case, and we should communicate a range of methodologies, including the procedural approach, which in a number of areas may offer a simpler and more elegant solution.

5. CONCLUSION

In teaching 'business' computing we may at times have neglected the technology underpinning those businesses. It is appropriate that we communicate programming styles that are relevant to, and appropriate in, a scientific environment. Specifically, students should be aware of the advantages gained through economy of expression, efficient code, and judicious use of a procedural approach.

REFERENCE

Ken Ritley, Scientific Computing in Java (Part 2): Writing Scientific Programs in Java, Gamelan http://softwaredev.earthweb.com/java/article/0,,12082_631281,00.html

```
int main( )
{
  double dPotentialEnergy, dKineticEnergy, dTotalEnergy;
  double dMassOfObject, dVelocityOfObject;
  double const dAccelerationOfGravity = 9.81;
  double dHeightOfObject;
  cout << "Please enter mass (kg), velocity (m/s) ";
  cin >> dMassOfObject >> dVelocityOfObject;
  cout << "Please enter height (metres) ";
  cin >> dHeightOfObject;
  dKineticEnergy = 0.5*dMassOfObject*dVelocityOfObject*dVelocityOfObject;
  dPotentialEnergy = dMassOfObject*dAccelerationOfGravity*dHeightOfObject;
  dTotalEnergy = dKineticEnergy + dPotentialEnergy;
  cout << "\nKinetic energy = " << dKineticEnergy;
  cout << "\nPotential energy = " << dPotentialEnergy;
  cout << "\nTotal energy = " << dTotalEnergy;
  getch();
  return 0;
}
```

Or this?

```
int main( )
{
  double PE, KE, E;       // potential, kinetic, total energy
  double m, v;            // mass, velocity
  double const g = 9.81;  // gravity
  double h;               // height
  cout << "Please enter mass (kg), velocity (m/s) ";
  cin >> m >> v;
  cout << "Please enter height (metres) ";
  cin >> h;
  KE = 0.5*m*v*v;
  PE = m*g*h;
  E = KE + PE;
  cout << "\nKinetic energy = " << KE;
  cout << "\nPotential energy = " << PE;
  cout << "\nTotal energy = " << E;
  getch();
  return 0;
}
```

Figure 1