ELECTRICAL AND INFORMATION ENGINEERING

University of the Witwatersrand, Johannesburg

Software Development II – Laboratory 4

Inheritance and Role Modelling

1 Introduction

This lab deals with inheritance, polymorphism and subtyping. These concepts form the cornerstone of object-oriented programming. In Section 3 a set of exercises is given which make use of an SFML graph plotter and illustrate inheritance being used for role modelling.

The laboratory outcomes are:

1. You understand `public` inheritance and when its use is appropriate.
2. You are able to extend existing inheritance hierarchies and create new ones; you are able to write client code which makes use of the base class of an inheritance hierarchy, and has no knowledge of the types that it is operating on.

As in the previous labs, one of the lab partners should download the starter code from the course website (the Laboratories section) and extract it to a local folder. Within this folder run `git init` to initialise the Git repo.

Now stage all of the untracked files and directories by typing: `git add .`
Commit the files to the *master* branch with: `git commit -m "Initial commit"`

Create a *solutions* branch on which to merge all the solutions for the exercises by typing:
`git branch solutions`

Share the repo with your partner by visiting GitHub and copying the **SSH** url from the *Quick setup* section of this lab's repo. Now in Git Bash, type:
`git remote add origin <paste SSH url here>`

Push your local branches to the remote by typing: `git push --all origin -u`

Now that the starter code, and the *master* and *solution* branches, are available in the shared GitHub repo, your lab partner needs to clone the remote repo using:
git clone <paste SSH url from GitHub here>

Once again, you will be creating a branch for the exercises in each section. Lab partners will alternate sections, so partner *A* will do all the exercises in Section 2 and partner *B* will do Section 3.

You are now in a position to start working on the lab exercises.

2 Introducing Inheritance — Payroll Calculations

Create, and checkout, a branch for this section by typing the following:
`git checkout -b section-2`

Remember to commit after each exercise, naming the commit correctly.

A company has various types of workers (modelled by the base class `Employee`) on its payroll. These include workers who receive a fixed weekly salary (modelled by the derived class `FixedRateWorker`) and workers who receive a salary based on the number of hours worked per week (modelled by the derived class `HourlyWorker`).

Exercise 2.1

The code for this section's exercises is contained in the `payroll` directory. View the source code in this directory and make sure you understand the inheritance hierarchy. Compile and run the main program by following the procedure described in Lab 3, section 1.

1. Answer the question that is asked with a comment in the `payroll-main` file.
2. Correct the source code in the inheritance hierarchy so that the output produced below the question matches that above the question.

Exercise 2.2

1. Create a new class *derived* from `Employee` that models the concept of a worker who receives a base salary plus a commission based on the number of items sold per week. Your class should allow a client to set the base salary, the commission per item sold, and the total number of items sold per week.
2. Write code in `main` to test that your new class works correctly.

Exercise 2.3

In the above exercises code and data is being reused within the `Employee` hierarchy. Specifically, the functions `first_name`, `surname` and `print` are inherited by all derived types and form part of the public interface of these types. The data members belonging to the `Employee` class (`_first_name` and `_surname`) are also inherited by the derived classes.

Inheritance facilitates this form of reuse but this should not be the primary reason for creating an inheritance relationship. Inheritance is more powerfully leveraged when we write code *which exists outside the hierarchy* but depends on the interface offered by the base class type. Clients of the base class do not have knowledge of the actual subtypes they are manipulating. This allows for the addition, removal and revision of these subtypes without requiring changes to the client.

> Inherit, not to reuse, but to be reused.

In order for this to work the code outside the hierarchy has to be programmed in terms of a pointer or reference to the base class of the hierarchy in question, and only call functions which form part of the base class's public interface. At run-time, the base class type is substituted with a derived class type and the polymorphic parts of the base class interface (the `virtual` member functions) are correctly resolved using dynamic binding.

1. Write a standalone function which merits the use of inheritance, polymorphism and subtyping in this section. Your function must calculate the total wages for the company workforce for a week. It should take a vector of `Employee shared_ptrs` and return the total wages for the week for all of the employees pointed to.
2. Test your function in `main` by passing it a vector containing pointers to each of the three types of employee that can exist.

Pull down the remote *solutions* branch to ensure that you have an up-to-date version. Now *merge* your commits on the *section-2* branch into the *solutions* branch. After doing this push your *solutions* branch back up to GitHub.

3 A Simple Graph Plotting Framework

Create, and checkout, a branch for this section by typing the following:
`git checkout -b section-3`

Remember to commit after each exercise, naming each commit correctly.

The program below plots a sine and cosine wave using a simple graph plotting framework.

```
int main()
{
   // setup Graph with Display
   const auto WIDTH = 800;
   const auto HEIGHT = 600;
   auto display = make_shared<Display>(WIDTH, HEIGHT);
   auto graph = Graph{display};

   // create sine and cosine functions
   auto amplitude = 1.0f;
   auto frequency = 1.0f;
   auto phase = 0.0f;
   auto sine_function = Sinusoid{amplitude, frequency, phase};
   auto cosine_function = Sinusoid{amplitude, frequency, phase + PI/2};

   // generate range and plot graphs
   auto range = Range{0, 6*PI};
   auto solid_red = SolidLineStyle{Colour::Red, display};
   graph.plot(generateDataPoints(sine_function, range), solid_red);

   auto solid_blue = SolidLineStyle{Colour::Blue, display};
   graph.plot(generateDataPoints(cosine_function, range), solid_blue);

   return 0;
}
```

Listing 1: Using the Graph Plotting Framework

All of the framework code is available in the `graph-plotter` directory. In its current form, the framework is quite limited and can only plot sinusoids using solid lines. The following exercises involve using, extending and modifying this framework.

Exercise 3.1

Build and run the executable for the graph plotter using CMake (follow the procedure described in Lab 2, section 5) and observe the output. Note, *you need to press the space bar* to update the graph, and the Escape key to quit the program.

Secondly, review the framework code which is contained in the `graph-plotter` directory. Ensure that you understand what the code is doing and how the framework is put together.

1. Go through each class and structure, identifying (with a comment above each definition) whether the class/struct belongs to the logic layer or to the presentation layer.
2. Explain (with a comment in `main.cpp`) why smooth sinusoid graphs are not produced by the framework, and make a simple change to correct this.

Exercise 3.2

1. Extend the framework so that it is capable of drawing sinusoids using both dashed lines (the dashes can be of any length) and dotted lines. This type of extension is well supported by the existing design and you should be able to meet these new requirements solely by writing new line styles — as opposed to modifying any existing code.
2. Change the `main` function to draw sinusoids using these new line styles.

Exercise 3.3

The framework currently only supports the plotting of sinusoids, which happen to be a particular type of function following the general form $y = f(x)$.

1. Modify the framework code so that any graph of the form $y = f(x)$ can be plotted, and ensure that it still works with sinusoids.
2. Extend the framework by modelling other functions which fit the general form, specifically:
   - Polynomials: $f(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_2 x^2 + a_1 x + a_0$ where $a_0, a_1, a_2, \ldots, a_n$ are constant coefficients.
   - Exponentials: $f(x) = Ae^{bx}$ where A and b are constant coefficients.

   Hint: functions contained in the `cmath` library might come in handy.
3. Test the functionality that you have added by changing `main` to plot:
   (a) $y = x^2 + 2x + 1$
   (b) $y = e^{1.5x}$

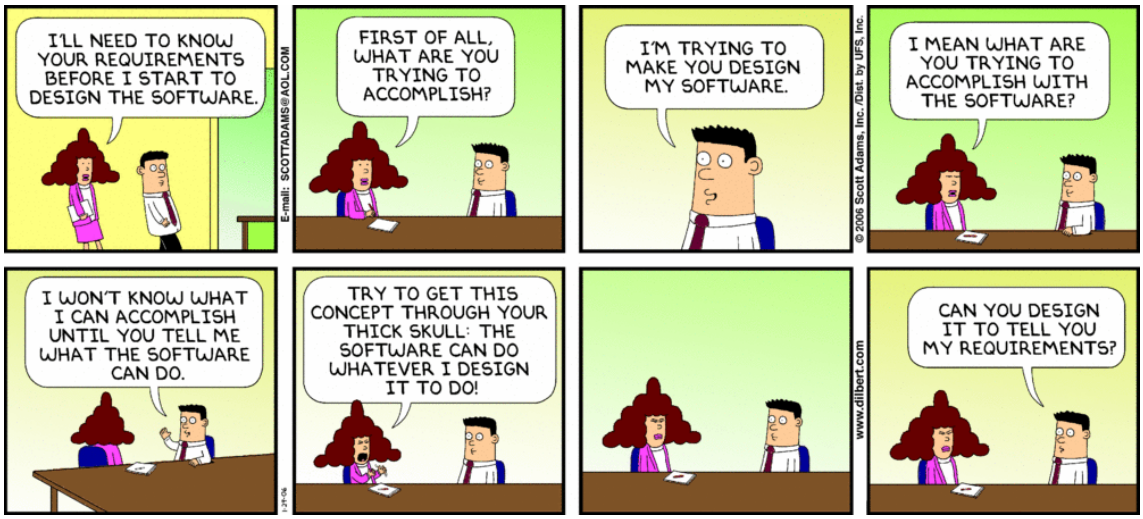
   for the range $-3 \leq x \leq 1.5$.

You can check the general shape of your plots by comparing them with plots generated by this online graph plotter.

Pull down the remote *solutions* branch to ensure that you have an up-to-date version. Now *merge* your commits on the *section-3* branch into the *solutions* branch. After doing this push your *solutions* branch back up to GitHub.

Once both partners have completed their section of the lab, submit it using a pull request on GitHub following the instructions in the Git/GitHub guide. The lab will be assessed according to the same criteria given in Lab 1. Your *solutions* branch must have a clean commit history.

Source: http://www.glasbergen.com/



Source: http://dilbert.com/strips/comic/2006-01-29/

Version 1.32, 2023 (elen3009-lab4.tex)