1 Objectives

This practical lab experience aims to achieve the following general learning objectives:

• To gain and consolidate some experience writing programs in Smalltalk, Eiffel and C++;
• To practically compare object-oriented programming features in Smalltalk and Eiffel;
• To examine two of the possibilities of implementing concurrency in C++.

2 Plagiarism Policy

The Department of Computer Science considers plagiarism as a serious offence. Disciplinary action will be taken against students who commit plagiarism. Plagiarism includes copying someone else’s work without consent, copying a friend’s work (even with consent) and copying material (such as text or program code) from the Internet. Copying will not be tolerated in this course. For a formal definition of plagiarism, the student is referred to http://www.ais.up.ac.za/plagiarism/index.htm (from the main page of the University of Pretoria site, follow the Library quick link, and then click the Plagiarism link). If you have any form of question regarding this, please ask one of the lecturers, to avoid any misunderstanding. Also note that the OOP principle of code re-use does not mean that you should copy and adapt code to suit your solution.

3 Submission Instructions

Upload all practical-related files as separate tar-gzip or zip archive (named s?????????.tgz, where ????????? is your student number), to the appropriate assignment upload on the course website. The deadline is Monday, 28 September 2015, at 17:00. The archive must include all the program source files that you have written.

4 Background Information

For this practical, you will be writing programs in Eiffel (Eiffel Studio 5.4 is available in the labs) and Smalltalk. You will have to compare these languages in terms of their support for different concepts related to object-orientation, inheritance and support for design by contract. You will also have to implement simple concurrent programs using both POSIX threads and OpenMP in C++. To achieve these tasks, you will have to write short programs to demonstrate how each language handles the concept under consideration.

The course website contains documentation related to Eiffel [2] and Smalltalk [1]. For the other languages, you are referred to the texts and references for previous courses related to the language.
5 Practical Tasks

This practical consists of the following three tasks:

Task 1: Smalltalk

For the Smalltalk component of this practical, make use of the GNU Smalltalk implementation (which is installed under Linux in the labs). You are required to implement a bank system in order to demonstrate the purely object-oriented design of Smalltalk.

Create an `account` class to act as a superclass for all accounts we will use to populate our bank. An `account` contains the instance variables `balance` and `holder`, which are a real and a string respectively. The `balance` is initially 1000. Furthermore, the following methods must be provided:

- `setHolder(String)` — sets the given account’s holder.
- `getHolder` — returns the name of the account’s holder.
- `getBalance` — returns the balance value of the account.
- `addBalance(real)` — adds an amount to the balance.
- `addInterest(real)` — applies an interest rate of 20% to the account.

Then create a number of subclasses to account (create at least 2) and create a new `addInterest` method for each account type. The kinds of accounts you create are up to you, but each must have a different interest rate.

The idea is that if a balance is positive, the `addInterest` method of the subclass is called. Conversely, if the balance is negative, the account superclass’ `addInterest` method is called instead, and therefore a different interest rate is applied when the account holder is in debt. You MUST use the `super` keyword to access the `account` base class to implement this functionality.

To tie this all together, create a `bank` class which contains an array to be filled with various accounts, and implements the following functionality:

- `addAccount(account)` — adds a new account subtype to the array
- `listAccounts` — prints a list of accounts to the screen
- `increaseBalance(String, real)` — attempts to increase the balance of the account that has a holder named specified by the first parameter, using the amount indicated by the second parameter
- `nextMonth` — calls the `addInterest` method of every account in the array

Task 2: Eiffel

You will have to create an animal farm simulator in Eiffel Studio. Create an `animal` class to act as a superclass for all animals, which you will use to populate the animal farm. The animal class will have to keep track of the animal’s name (represented as a string) as well as its hunger level (represented as an integer). The hunger will initially be zero, indicating that the animal is well-fed, and will go up to a maximum of 5, indicating that the animal is starved. The animal class will also need the following methods:

- `setName(string)` to set the name of the animal.
- `getName` to return the name of the animal.
- `getHunger` to return the hunger level of the animal.
- `feed(string)` which will reduce the animal’s hunger level by one, where the string describes the food.
- `sleep` which will increase the animal’s hunger level by one.

You will then have to create at least 2 subclasses for the `animal` class. In these subclasses, you will have to override the `feed` method to only accept a certain kind of food. If another type of food is fed to it, the hunger level should not decrease. You may decide on your own animals and preferred food types (for example, a wolf might prefer meat, while a lamb might prefer grass). The `feed` method should produce output describing the animal, the type of food it is being fed with, and the whether it eats the food or not. The `sleep` method should produce output describing the animal, and the fact that it has slept.
You will also need a `farm` class to manage the animals. This class will keep a list of animals. You will need to add methods for adding and removing animals, feeding a particular animal in the list with its appropriate food type, and a method to call the `sleep` method of each animal.

Eiffel provides syntax for implementing classes with a “Design by contract” methodology. Use the following assertions in your program:

- Add a class invariant that will ensure that the animal’s hunger level is always between 0 and 5 (inclusive).
- Add a requirement to each animal’s `feed` method to ensure that it is being fed the correct food.
- Add a variant to the loop that calls the `sleep` methods, which decreases after each iteration, down to zero, so that the loop is guaranteed to terminate.
- Add a loop invariant to the same loop, which ensures that each animal has a valid hunger level through each iteration of the loop.

Task 3: Concurrency

For this part of the practical, you must make use of C++11 standard library threads and OpenMP in C++. Your programs must read a candidate “magic square” of values from a file (an example file will be provided on the course website; The first number in the file indicates the width and height of the square).

A “magic square” is a square matrix of values, in which the values in each row, column and diagonal in the matrix add up to the same value. For example, the following matrix is considered a “magic square”:

```
2  7  6
9  5  1
4  3  8
```

Your programs must calculate the total of each column, row and both diagonals in parallel (i.e., each column will be handled by a separate thread, as will each row and each diagonal. All these threads must execute at the same time). This will effectively produce an array of total values, where each thread will only store its total in a separate space in the array. For simplicity, you may assume that the matrix is a $3 \times 3$ grid.

Furthermore, your programs must count the total number of even values in the square. All the threads in your programs must use the same variable to accumulate the count of even values. If the threads simply accumulate the count in a thread-local variable and then add these counters up at the end of execution, you will be penalised.

Once these calculations are done, your program must print out whether the square is a “magic square” or not, and the number of even values that were found. You must also ensure that the program is thread-safe.

This means that even if the program has executed correctly, we will evaluate the code to see if you took precautions to prevent the problems commonly associated with multi-threaded programs. This includes race conditions and the possibility of the final messages being printed out before data processing is done.

You must write 2 programs. The first must make use of standard library threads provided in the C++11 standard. For this your program must include the `<thread>` header. You will also need to add the relevant arguments to the compilation command you use. For example:

```
g++ -std=c++11 -pthread yourProg.cpp
```


The second program must make use of OpenMP. For this your program must include the `omp.h` header. You will also need to add the relevant compile arguments. For example:

```
g++ -fopenmp yourProg.cpp
```

A tutorial can be found at [https://computing.llnl.gov/tutorials/openMP/](https://computing.llnl.gov/tutorials/openMP/).
6 Marking

Both the implementation and the correct execution of the programs will be taken into account. Marking will take place in the lab sessions during the week of 28 September 2015. Also ensure that you upload your implementations into the assignment upload slot. Each of the programs will count 5 marks for a total of 15 marks, as follows:

<table>
<thead>
<tr>
<th>Category</th>
<th>Mark Allocation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eiffel</td>
<td>5 marks</td>
</tr>
<tr>
<td>Smalltalk</td>
<td>5 marks</td>
</tr>
<tr>
<td>Concurrency</td>
<td>5 marks</td>
</tr>
</tbody>
</table>

References
