SCHOOL OF SOFTWARE ENGINEERING OF USTC

FIRST SEMESTER, 2012–2013
Campus: Suzhou

AGILE SOFTWARE DEVELOPMENT

Reusable modules and polymorphism in *C*

(Time allowed: TWO hours)

The objective of this experiment class is to put to practice the teachings from the lecture on structured *C*, where it is shown that is perfectly possible to have objects in *C*. Also, the simple object polymorphism approach introduced during the lecture will be used.

The topic is the implementation of a very simplified virtual machine. We want to have a virtual machine that is easy to extend with new instructions. All the work should be done in pure *C*. This is the part one of this work, the other part will be next week, focused on error handling techniques.

You have to work alone, one student per computer, this is non-negociable. It is to help you to be focused and practice by yourself. If you are stuck after trying various ways, you can call for help. You might have to patient while the teacher is busy helping someone else.

1. For our virtual machine, we will need to implement a reusable *stack* object. We will consider that the stack contains *double* values, and only that. The stack can grow infinitely. We want the following methods:

- *init*, setup an empty stack
- *destroy*, release all the memory used by the stack
- *clear*, empty the stack
- *size*, return the size of the stack
- *push*, push one *double* value to the stack
- *pop*, pop the top value from the stack and return it
- *top*, return the top value of the stack
- *dump_state*, prints the state of the stack in human-readable fashion

A friendly advice : before moving out the next questions, consider testing your code seriously. Do some stack manipulation, and check that at each step, the stack state is what it should be.

2. We will now define the instructions of our virtual machine. One instruction take as input a stack, and will modify its state.

- *add*, pop a and b, push $a + b$
- *sub*, pop a and b, push $a - b$
- *mul*, pop a and b, push $a \times b$
- *div*, pop a and b, push $\frac{a}{b}$
- *sqrt*, pop a, push $\sqrt{a}$
- *value*, push a given value

CONTINUED

- *close*, pop a and b, push 1 if $|a - b| \leq (\alpha_{tol} + |b|\beta_{tol})$ else push 0. This instruction is to check if two floating point values are close. α_{tol} and β_{tol} are two parameters of the instruction. By default, a sensible choice would be $\alpha_{tol} = 10^{-8}$ and $\beta_{tol} = 10^{-5}$.

We want an *instruction* object, and the 7 instructions would be subtype of this object. This way, the code of each instruction is well separated from the others. Adding new instructions does not impact on the other instructions. Thus, the instruction object would have the following methods.

- *destroy*, release all the memory used by the instruction
- *do*, taking a *stack* as only parameter and compute an operation.
- *get_name*, return the name of the instruction

Implement the *instruction* object and its seven subtypes, using the solution shown during the lecture, ie. using a delegate structure to store function pointers. Take care of *all* the memory you used.