CSE 332 Studio Session on Design Patterns in C++

Design patterns provide a vocabulary for identifying and describing both common design problems and known reusable approaches to solving them. In this studio you will gain experience applying design patterns relevant to several common design problems that arise in C++ programming, at an introductory level. Each design pattern has (at least) a name, a description of the problem, a solution description, and contextual factors (such as the pattern’s intent) to consider when selecting or applying the pattern. More details about design patterns are at [http://en.wikipedia.org/wiki/Design_Patterns_(book)](http://en.wikipedia.org/wiki/Design_Patterns_(book)) (the patterns are linked at the bottom of the page) and [http://en.wikipedia.org/wiki/Design_pattern_(computer_science)](http://en.wikipedia.org/wiki/Design_pattern_(computer_science)).

In this studio you will again work in small groups, as before students who are more familiar with the material are encouraged to help those for whom it is less familiar, and asking questions during the studio sessions is highly encouraged as well.

Please record your answers you work through the following exercises. After you have finished please send your answers to the required exercises, and to any of the enrichment exercises you completed, in an e-mail to the cse332@seas.wustl.edu course e-mail account, with the subject line “Design Patterns Studio”. The enrichment exercises are optional but are a good way to dig into the material a little deeper, especially if you breeze through the required ones. Please make sure as you work through these exercises that each team member has a chance to participate actively — e.g., take turns coding, looking up details, debugging, etc., and please also refer to the slides and the posted code examples as you work.

PART I: REQUIRED EXERCISES

1. Find your team members in the studio area, sit down at/around and log in to one of the Windows machines, open up Visual Studio and create a new Visual C++ Win32 Console Application project. Change the signature of the main function in the source file that Visual C++ generated to match the one that was specified for the previous studios. Write down the names of the team members who are present (please catch up anyone arriving late on the work, and add their name) as the answer to this exercise.

2. (Iterator pattern). Declare an Animal struct, with a const pure virtual print method that takes a reference to an ostream and has a void return type. Derive (via public inheritance) a Giraffe struct and an Ostrich struct from the Animal struct, and override the virtual print method so that it prints out a message indicating the specific kind of animal it is (Giraffe or Ostrich, respectively). In your main function, declare a Giraffe object and an Ostrich object on the stack, and push their addresses back into a vector of pointers to Animals. Write a for loop that uses the vector’s [] operator to call the print method on each of the objects whose address is stored in the vector, passing cout to each call. Build your solution, fix any errors or warnings, and run your program to make sure the right information is being printed out. As the answer to this exercise, describe (1) how both this solution and an alternative that uses an STL iterator (which you’ll do in the next exercise) are examples of applying the Iterator pattern, and (2) for which kinds of STL containers the solution used in this exercise would work, vs. when you would need to use the STL iterator version.
3. **(Factory Method pattern)**. Move the for loop from the previous exercise into a print function template that has a single parameterized type (for the type of container it takes), takes a reference to an object of the parameterized type and a reference to an ostream, and has a void return type. Modify the for loop so it uses an STL style iterator (of the appropriate type associated with the parameterized container type) and again calls the print method on each object whose address is in the container (with the ostream reference).

In your main program, add a list of pointers to animals, and push back the addresses of the Giraffe and Ostrich objects from the previous exercise into it. Call the print function template with this list and cout, and also with the vector from the previous exercise and cout. Build your solution, fix any errors or warnings, and run your program to make sure the right information is being printed out.

As the answer to this exercise, describe how (a generic programming variation of) the Factory Method design pattern is used in this solution to make it work generically for both vectors and lists of pointers to animals.

4. **(Singleton pattern)**. Declare a Zoo class, with a public static instance method that takes no parameters and returns a pointer to a Zoo object, and a public (non-static) add method that takes a pointer to an animal and pushes it back into the list member variable described below. To ensure only a single instance, the Zoo class should declare and define a private default constructor that prints out a message indicating that a zoo has been created. To prevent copying, the Zoo class should declare but not define a private copy constructor and a private assignment operator. The Zoo class should have a private list of pointers to animals and a private static pointer to a zoo object, as member variables.

Outside the zoo class, define the static pointer member variable and initialize it to 0. Define the static instance method so that if (and only if) the static member variable is 0, it dynamically allocates a single zoo object and stores the address of the newly allocated object in the static member variable. In either case (whether or not the static member variable was 0 when the static member function was called) the static member function should then return the value of the static pointer member variable.

Define (and declare as a friend in the zoo class) an insertion operator (operator<<) that takes a reference to an ostream and a reference to a const zoo object, and uses the print function template to print out the contents of the zoo’s list of animals, using the ostream.

In your main function, remove the lists and vectors from the previous exercises and instead add the addresses of the ostrich and giraffe objects to the singleton zoo object, using its static instance method to obtain its address and then call its non-static add method using that address. Also insert the singleton zoo object (**hint**: you’ll need to dereference the address returned by the static instance method) into cout to print out the animals you added. Build your program, fix any errors or warnings, and run it to make sure the correct output was produced.

As the answer to this exercise, show the output of your program, and explain how you can tell from it that a single zoo object was used throughout the program’s execution.
5. **(Visitor pattern).** Derived classes in an inheritance hierarchy often have distinct methods and behaviors, which only apply to their concrete types. A simple (but useful) version of the handshaking used by the Visitor design pattern can be realized using dynamic casting of pointers to a base type.

Add a `look` method to your giraffe class that prints out a message that the giraffe is looking around and can see far away. Add a `hide` method to your ostrich class that prints out a message that the ostrich has buried its head in the sand and cannot see anything. In your zoo class, add an `observe` method that tells what each animal in the zoo is doing: it should do a dynamic cast of each pointer to a pointer to the concrete type of animal (ostrich or giraffe) and if the resulting pointer is non-zero use it to invoke the appropriate concrete method (look for the giraffe, hide for the ostrich). Call that observe method on the single zoo object, from your main function. Build and run your program, fixing any errors or warnings you encounter. As the answer to this exercise, give the output that was produced by your program.

6. **(Prototype pattern).** Add a public pure virtual `clone` method that takes no parameters and returns a pointer to an Animal, to your Animal struct. Override that method (as virtual rather than pure virtual) in your derived Giraffe and Ostrich structs so that a new instance of the appropriate type (Giraffe or Ostrich respectively) is dynamically allocated, initialized by copy construction (with a reference to the object on which the clone method was invoked passed as the copy constructor’s only parameter), and its address then returned by the clone method.

Make the copy constructor of your zoo class public, and define it using calls to the clone method given above to make a deep copy of the animals in it. Also add a Boolean flag to your zoo class that indicates ownership of the animal objects in it: set the flag to true in the copy constructor, and to false in all other constructors. In the zoo class destructor, check for that ownership flag being true and if (and only if) it is true then iterate through and call delete on all the pointers to animals in that zoo object.

In your main function, after the place where the Ostrich and Giraffe objects were added to the singleton zoo object, copy construct another zoo object on the stack (using a reference to the singleton zoo object as the parameter to the copy constructor). Print out the contents of both the stack zoo object and the singleton zoo object and as the answer to this exercise indicate whether or not they match (they should).
PART II: ENRICHMENT EXERCISES (Optional, but if doing both complete 7 before 8)

7. **(Memento pattern).** Starting with the code from the previous exercise, add to your Animal struct a pure virtual `memento` method that takes no parameters and returns a string. In your Giraffe and Ostrich classes override this method so that it returns a string representing what type of animal it is (for example “G” for Giraffe or “O” for Ostrich). Add to your zoo class a `memento` method that takes no parameters and returns a string containing the types of animals in the zoo, separated by spaces (this is easy to implement by calling the memento methods of the animals in the zoo. Add to your zoo class a static `restore` method that takes a string in the format produced by the `memento` method, uses the instance and add methods from the previous exercises to add animals corresponding to those represented by the string, and sets the ownership flag to true. Also add to your zoo class a static `destroy` method that calls delete on the static pointer member variable and then sets it to zero.

In your main function after the point where you have printed out the contents of the singleton zoo object, use the memento method to obtain a string representing the contents of the singleton zoo object, call the destroy method, pass the string into the restore method and then print out the contents of the singleton zoo object. As the answer to this exercise indicate whether or not the contents of the singleton zoo object before and after the memento/destroy/restore calls match (they should).

8. **(Adapter pattern).** A variation of the Adapter pattern can be used to introduce polymorphic relationships among unrelated objects (for example, if you don’t have the ability to modify the classes directly) by wrapping them in interfaces that are related by inheritance. This variation of the Adapter pattern is called the External Polymorphism (see [http://www.cs.wustl.edu/~schmidt/PDF/External-Polymorphism.pdf](http://www.cs.wustl.edu/~schmidt/PDF/External-Polymorphism.pdf) for more details), and offers a capability similar to the Java notion of an interface.

Starting with the code from the previous exercise, declare an animal adapter struct that has a const pure virtual `print` method that takes a reference to an ostream and has a void return type (you could use multiple inheritance to extend the animal struct from the previous exercises, but we won’t here) and a const pure virtual `behave` method that takes a reference to an ostream and has a void return type. Derive concrete ostrich adapter and giraffe adapter structs from the animal adapter struct, each of which has a constructor that takes a pointer to an object of the concrete type (ostrich vs. giraffe, respectively) and stores it in a member variable of the appropriate concrete pointer type. Implement the adapters’ `print` methods by simply calling the print method of the concrete object that is being adapted. Implement the adapters’ `behave` methods by calling the look method if the wrapped object is a giraffe, or the hide method if the wrapped object is an ostrich.

Modify your zoo so that it stores pointers to the animal adapter struct rather than pointers to the animal struct. Modify its add method (and any other methods as needed) so that it uses a dynamic cast to identify the kind of animal being passed to it, create and store the address of an appropriate concrete adapter object for that kind of animal (ostrich vs. giraffe), and uses the polymorphic behavior of the adapter to replace the visitor pattern approach you implemented in the previous exercise (the dynamic casts should all move to the add method, and all other methods should rely on the polymorphic adapter interfaces).

Build your program, fix all warnings and errors, and run your program. As the answer to this exercise, give the output your program produced, and confirm that the same output was produced as in the previous exercise.