

CSE687 Midterm #4

Name: _____ Instructor's Solution_____

This is a closed book examination. Please place all your books on the floor beside you. You may keep one page of notes on your desktop in addition to this exam package. All examinations will be collected promptly at the end of the class period. Please be prepared to quickly hand in your examination at that time.

If you have any questions, please do not leave your seat. Raise your hand and I will come to your desk to discuss your question. I will answer all questions about the meaning of the wording of any question. I may choose not to answer other questions.

You will find it helpful to review all questions before beginning. All questions are given equal weight for grading, but not all questions have the same difficulty. Therefore, it is very much to your advantage to answer first those questions you believe to be easiest.

1. Discuss the importance of inheritance as implemented in C++. Enumerate the coding errors presented in class that interfere with proper operation of inheritance hierarchies.

Answer:

Inheritance supports code reuse by sharing base class code with all its derived classes. More important though is its enabling of substitution. Any client's Base class pointer can bind to derived instances and the client can use the pointer to invoke derived class functions using the base protocol.

Here is a list of design errors that cause unexpected behavior of polymorphic operations or failures to compile:

- a. **Overloading base class functions in derived classes.** Overloading across class scopes is a derived class error that hides all the other base overloads.
- b. **Overloading base class virtual functions.** This a base class error that causes correctly defined overrides of the overloaded base virtual function to hide the other base overloads of that virtual function.
- c. **Redefining in a derived class non-virtual base class functions.** This is a derived class error that causes invocations of the non-virtual method with a base pointer bound to the derived class instance to call the base function, not the redefined derived class function.
- d. **Providing default values for parameters in base classes may be inconsistent with default values set by derived classes.** The choice of which default to use is based on the type of the client's pointer or reference not the bound type.
- e. **Failing to provide a virtual destructor** in any class that may be a base for derivation is an error that will cause incomplete destruction of a derived instance created on the native heap and accessed through a base pointer. Only the base destructor will be called.
- f. Derived class methods should require the same or weaker specifications on the method inputs than the base class specification. They should require the same or stronger contracts on method outputs than the base method outputs satisfy. Failing to do so may cause behaviors inconsistent with the base methods.
- g. Any class derived from one or more bases and containing composed instances that provides definitions for constructors **must initialize all its bases and composed instances in constructor initialization sequences.** Otherwise incorrect construction may occur.
- h. When using multiple inheritance the **use of public virtual inheritance** to merge common code in multiple bases can cause errors because the common code is initialized by the most derived class. But that may break initialization needed for the most derived's bases.

2. The `std::for_each` STL algorithm iterates over elements of a collection, applying a callable object to each element. Write a lambda that appends an input string to each element of a `std::list<std::string>`. Show all the code to initialize a test list with a few strings, create the lambda, and apply it to the list using the `std::for_each` algorithm. Here is the signature of the `for_each` function:

```
template <class Iter, class UnaryFunction>
UnaryFunction for_each(Iter first, Iter last, UnaryFunction f)
```

Answer:

```
std::list<std::string> theList{ "one ", "two ", "three", "four " };
std::string appendString = " - element";
std::for_each(
    begin(theList),
    end(theList),
    [&appendString](std::string& elem) { elem += appendString; }
); // elem is std::string element of the list supplied by std::for_each(...)
    // when it invokes the callable object, e.g., the lambda

for (auto& elem : theList)
{
    std::cout << "\n " << elem;
}
```

3. What is the purpose of the AbstractXmlElement in the XmlDocument demo code provided as a helper for Project #2. Cite, for each of the Design Principles discussed in class in lectures last week, how they apply to Project #2, or if not, state that they do not apply.

Answer:

The AbstractXmlElement class has two roles:

- a. Provide a communication contract for clients to use.
- b. Share some method code with derived classes.

Liskov Substitution Principle:

Most of the member functions of the XmlDocument and XmlElement classes use `std::shared_ptr<AbstractXmlElement>` smart pointers to access and operate on document elements. These bind to instances of any class derived from AbstractXmlElement and support LSP.

Open-Closed Principle:

Behaviors of the derived classes can be extended by overriding virtual base methods. Also, global functions like DFS can accept callable objects to operate on each element of a document. Thus the document behaviors can be extended by defining additional callable objects.

Dependency Inversion Principle:

DIP is supported in that clients use the AbstractXmlElement abstraction and makeElement functions without binding to specific derived classes. Any time they directly create or manipulate one of the derived instances they are no longer using DIP.

Interface Segregation Principle:

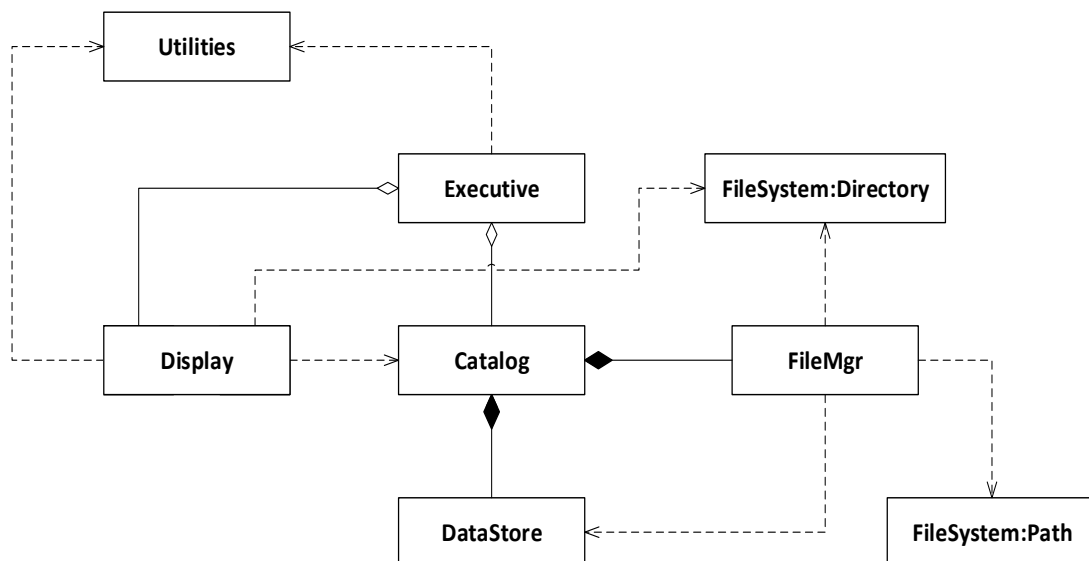
One could argue that the ISP is used because the XmlDocument functionality is divided between the Document and Element classes, each with interface focused on the functionality they supply. You could also argue that it doesn't apply since some of the derived classes in XElement are forced to depend on methods they don't use. Since one could argue about this I will accept either answer if you say something sensible about why you gave that answer.

4. Draw a class diagram for your design of Project #1. Discuss all relationships between classes in your design and make sure that the class diagram reflects them.

Answer:

The Executive aggregates instances of Display and Catalog. It passes its commandline and a pointer to the Catalog instance to Display in its constructor arguments. Catalog composes instances of DataStore and FileMgr. It passes a reference to DataStore to FileMgr in a FileMgr constructor initialization sequence. FileMgr uses Catalog's DataStore to save file and path information. Catalog then makes that information available to display through a set of member functions.

Both Display and FileMgr use FileSystem through invocations of its static member functions. DataStore and Executive use Utilities global title(...) function to create the project's output stream. Executive and Display use Utilities through its global function title. This is simply a package¹ containing one global function and so usually would not be shown on a class diagram. I bend the rules a little bit when that helps reader to understand the code.



¹ A package with only global functions is like a class of which there is only one instance.

5. Write a class declaration for a logging facility that accepts a message that includes a date-time string. Each message includes text describing the logged event and may contain other parts, unspecified at this time. Instances of the class must store multiple messages in memory and may be requested to save all logged messages to a file and clear the internal message store and be ready to continue logging after the save operation. How would you configure the class so that only one instance of the class could be created?

Answer:

```
template <typename T>
class Message
{
public:
    Message(const std::string& timeDate, const std::string& test, const T& t);
    void show();
    std::string toString();
private:
    std::string timeDate_;
    std::string text_;
    T t_; // here are those other unspecified parts
};

template <typename T> // You were not expected to provide an interface and factory.
struct ILog          // I did that to provide a solution for another exam.
{
    virtual ~ILog() {}
    virtual void message(Message<T>& msg) = 0;
    virtual void show() = 0;
    virtual bool save() = 0;
};

template <typename T>
struct factory
{
    ILog<T>* create();
};

template <typename T>
class Log : public ILog<T>
{
public:
    void message(Message<T>& msg); // adds message to log
    void show();                  // shows log messages
    bool save();                  // saves to file and clears log
private:
    static std::vector<Message<T>> log_; // this ensures that only one log
                                        // can be created
};

template<typename T>
std::vector<Message<T>> Log<T>::log_;

template <typename T>
ILog<T>* factory<T>::create() { return new Log<T>(); }
```

6. Show how you would add a Depth First Search member function DFS(...) to the XmlDocument class that accepts a callable function to operate on each derived element encountered during the search². You don't need to write the code for DFS(...), just show how it is declared in the class. Show how you would declare the callable object – you don't need to provide its internal code.

Answer:

```
class XmlDocument
{
public:
    using sPtr = std::shared_ptr < AbstractXmlElement > ;

    -- many methods elided --

    template<typename CallObj>
    void DFS(sPtr pElem, CallObj& co); // operates on derived element *pElem
private:
    // with callable object co
    sPtr pDocElement_;           // AST
    std::vector<sPtr> found_;     // query results
};

//-----< search subtree of XmlDocument >-----

template<typename CallObj>
void XmlDocument::DFS(sPtr pElem, CallObj& co) // you don't need to
{                                              // include this implementation
    co(*pElem);
    for (auto pChild : pElem->children())
        DFS(pChild, co);
}

std::function<void(AbstractXmlElement&)> f;

f = [] (AbstractXmlElement& Elem) {
    if (Elem.tag().size() > 0)           // you don't need to show
        std::cout << "\n " << Elem.tag(); // these two lines
};

doc.DFS(doc.xmlRoot(), f);
```

² You don't need to show all the member declarations for XmlDocument, only the class declaration with the DFS member. Thus you may elide all the other members [elide means to remove].

7. When you design the base class for an inheritance hierarchy what control do you have over the methods provided by derived class designers?

Answer:

- a. If you make a base function non-virtual no derived class should redefine it.
- b. If you make a base function virtual derived classes may override it.
- c. If you make a base function pure virtual derived classes must redefine it.