

CSE687 Midterm #2

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. Write all the code for a Descendents(const std::string& tag) member function for the XmlDocument class, from the Project #2 helper code, that finds all of the descendants of the first element found, with tag name equal to the input tag string, in Depth First Search order. What should you do if the tag is not found? How will you return the results? You may assume that the class hierarchy contains a member function Element(const std::string& tag) that returns a smart pointer to the first element found with that tag.

Answer:

A fully correct answer provides the equivalent of the find and descendents methods. I've included a little more to show you the context of my solution.

```
using sPtr = std::shared_ptr < AbstractXmlElement > ;

//----< find element(s) with this tag >-----
/* if tag == "" returns pElem and all descendents */
bool XmlDocument::find(const std::string& tag, sPtr pElem, bool findall)
{
    if (pElem->tag() == tag || tag == "")
    {
        found_.push_back(pElem);
        if (!findall)
            return true;
    }
    for (auto pChild : pElem->children())
        find(tag, pChild);
    return (found_.size() > 0);
}
//----< find element with this tag >-----
/*
 * found_[0] contains first element (DFS order) with tag, else empty
 */
XmlElement& XmlDocument::element(const std::string& tag)
{
    found_.clear();
    find(tag, pDocElement_, false);
    return *this;
}
//----< find all descendents of last find with this tag >-----
/*
 * returns all descendents if tag == ""
 */
XmlElement& XmlDocument::descendents(const std::string& tag)
{
    if (found_.size() == 0)
        found_.push_back(xmlRoot());
    sPtr pElem = found_[0];
    found_.clear();
    for (auto pChild : pElem->children())
        find(tag, pChild, true);
    return *this;
}
//----< return found results >-----

std::vector<std::shared_ptr<AbstractXmlElement>> XmlDocument::select()
{
    return std::move(found_); // returns results and clears found_
}
```

2. How do static member functions and static member data of a class differ from non-static members for that class?

Answer (phrases in bold are most essential characteristics):

Values of static member data are:

- a. stored in static memory and are valid for the lifetime of the program.
- b. **shared by every instance of the class** in which they are defined.

Static member functions:

- a. can be invoked using the class name.
- b. **can only access static member data** of the class.

Values of non-static member data are:

- a. stored with the instance and are valid for the lifetime of the instance.
- b. **unique** to each instance.

Non-static member functions:

- a. are invoked using the name of the instance on which they operate.
- b. **can access both static and non-static member data** of the same class

3. When is it appropriate to supply a move constructor for a class you are designing? Describe the things a move constructor needs to do. You do not need to supply code for the answer, but may do so.

Answer:

Note: I inadvertently gave an incorrect signature for move constructor and move assignment in the C++11 design note, using `const` argument. The argument should **NOT** be **`const`**. If you repeated that error I marked it as incorrect but did not deduct points for that.

Note: A move constructor usually does not move data, it moves pointers to data.

It is appropriate to supply a move constructor, `X::X(X&& x)`, when any of the following are true:

- a. Instances of the class hold pointer(s) to resources stored on the native heap and copying pointers to the resources is significantly cheaper than copying the resources. A move constructor creates the new instance then copies the pointer(s) held by the source instance to its own pointer(s). It then sets the source pointer(s) to `nullptr`.
For STL containers the move swaps the container's resource pointers and clears the source, so we have essentially transferred ownership of the container's state.
- b. The class holds bases and members all of which are moveable. The move constructor needs to use `std::move(...)` in the initializer sequences for bases and members.
- c. Class resources can be moved but not copied, as is the case for `std::unique_ptr<T>`.
- d. For classes that hold unique ownership to some resource where we may need to transfer ownership. A socket class is an example of such a case (we'll discuss that when we talk about sockets). In this case 4 bytes of data are copied. No pointers are involved.

For the case d., above, the move constructor may copy the resource composed by the source and supply a value for the source signifying invalid, `INVALID_SOCKET` for example.

Move constructors are called when passing an rvalue (temporary) by value or when we use the `std::move(...)` to cast an lvalue to an rvalue, telling the compiler we won't use the source object again.

If copying presents no performance issues and transfer of ownership is not needed then we don't need and should not provide move operations.

4. Why would you use an interface and object factory in one of your designs? What benefits do you gain by doing that?

Answer:

(Your grade for this question is determined in large part by how clearly and completely you explain the benefits and how they are achieved.)

A component is software that provides an interface¹ and object factory² for all client access. We try to build our components as independent entities that can be combined with other components to build complex software systems.

When a client uses an interface pointer, bound to some object that implements the interface, the invocation signature depends on the interface, not on the names of concrete types used by the implementer.

The separation of client from the interface implementer is completed by making available an object factory for the client to call to create instances of the interface implementer.

Thus, no matter what changes are made to the implementer the text of the client does not change as long as the component interface and object factory interface do not change³. And that means that the client does not have to recompile when the implementer changes. If you build the component as a DLL the clients don't even have to relink as the loader resolves references into the DLL at load time.

Often you use interfaces and object factories for library components that a lot of code uses and the library may be under active maintenance. You may also use them for parts of a developing system that are undergoing intense development activity and so are subject to change. This stops propagation of change throughout a system, keeping it contained inside the component.

¹ A C++ interface is a class or struct with no data members, no constructors, and all pure virtual functions, with the exception of a virtual destructor with empty body.

² An object factory is a class or global function that creates instances of types defined in the component's implementation.

³ Changes to component implementation may require changes in the factory implementation, but not to its interface so the client doesn't see those changes.

5. Write all the code for a class that holds a collection of smart pointers to instances of an arbitrary type on the heap. Write a function that accepts a callable object to perform operations on each heap instance. Assuming that the instances provide a function with signature `void say()`, write a callable object that invokes the `say()` function on each instance with the help of the accepting function.

Answer:

```
class Verbal
{
public:
    Verbal();
    void say();
private:
    static size_t numberOfInstances_;
    size_t myNumber_;
};
size_t Verbal::numberOfInstances_ = 0;

Verbal::Verbal() : myNumber_(++numberOfInstances_) {}

void Verbal::say()
{
    std::cout << "\n Verbal instance #" << myNumber_;
}

template<typename T> // answer starts here -----
class Collection
{
public:
    using iterator = typename std::vector<std::unique_ptr<T>>::iterator;
    iterator begin() { return collection_.begin(); }
    iterator end() { return collection_.end(); }
    void add(T* t);
private:
    // collection of smart pointers to instances of an arbitrary type
    std::vector<std::unique_ptr<T>> collection_;
};

template<typename T>
void Collection<T>::add(T* t)
{
    collection_.push_back(std::unique_ptr<T>(t));
}

template<typename CallObj>
void Operation(Collection<Verbal>& c, CallObj& co) // function taking callable object
{
    for (auto& pElem : c)
    {
        co(*pElem); // operating on each heap instance
    }
    // smart pointers should always point to instance on heap
}

The second argument, below, is the callable object:

Operation(c, [](Verbal& v) { v.say(); }); // answer ends here -----
```

6. State the Liskov Substitution Principle and describe where you have or will use it in Project #2.

Answer:

"Functions that use pointers or references statically typed to some base class must be able to use objects of classes derived from the base through those pointers or references without any knowledge specialized to the derived classes."

Almost all of the member functions of the `XmlElement` and `XmlDocument` classes use LSP. They refer to elements they use through `AbstractXmlElement` pointers, invoking their functionality without any knowledge of the specific bound type.

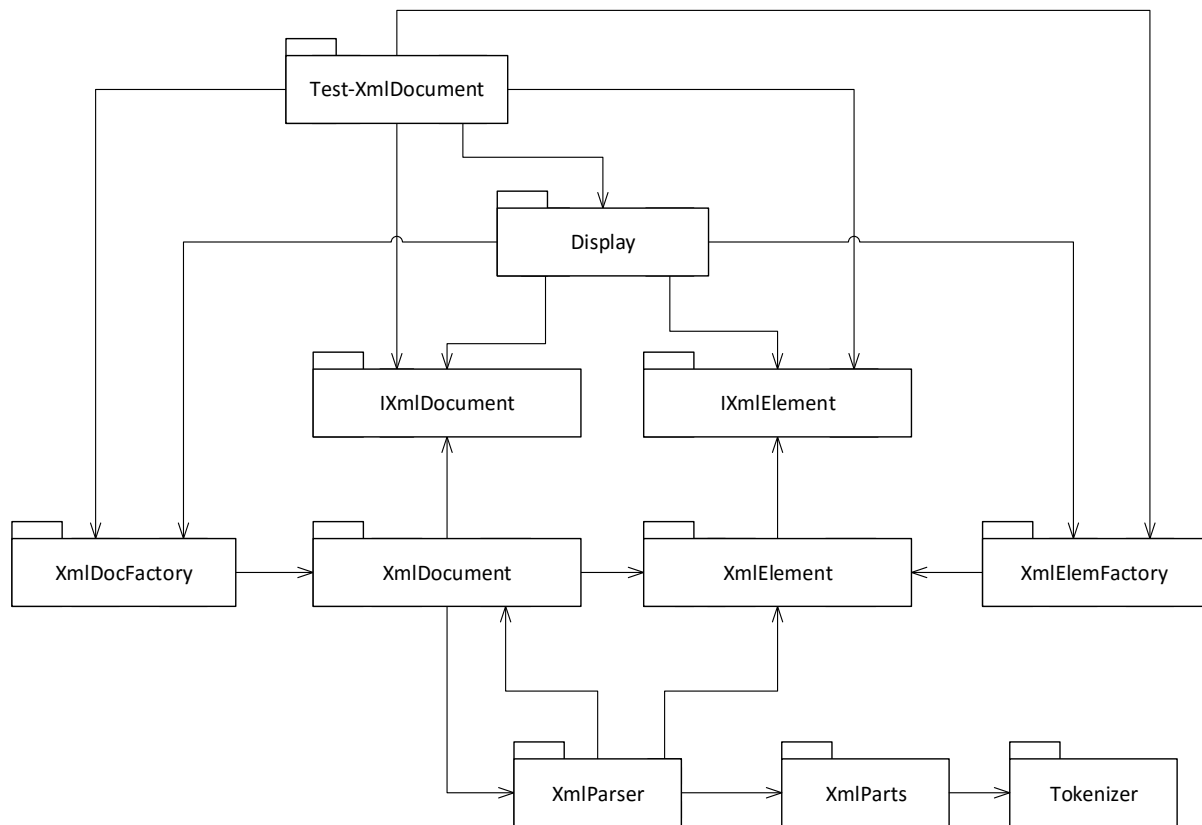
There are also global functions that use `AbstractXmlElement` pointers. These pointers are `std::shared_ptr<AbstractXmlElement>` instances which preserve the substitutability⁴ of native `AbstractXmlElement*` pointers.

⁴ That's because the smart pointer code uses the underlying native `AbstractXmlElement*` pointer for all its reference operations.

7. Draw a package diagram for your design of Project #2. Assume that you want to allow code that uses the XmlDocument to include a header file with no implementation detail and load the XmlDocument functionality as a dynamic link library. Your diagram should include all the code you would submit with the project.

Answer:

I've shown interfaces, XmlDocument and XmlElement, and object factories, XmlDocFactory and XmlElemFactory, residing in separate packages. It's likely that, for user convenience, we would merge those into one package so clients only have to include a single header file. This is the header file with **no implementation detail** (that must be interface(s)).



Note on flows:

- The dependencies between **XmlDocument** and **IXmlDocument** and between **XmlElement** and **IXmlElement** are due to inheritance. All others are calling relationships.
- Many of you had insufficient detail or inaccuracies in the package flows.

Note on superfluous parts. We don't need:

- XmlWriter** - the element `toString()` member function is all that we need to write out the **XmlDocument**, as demonstrated in class.
- FileMgr** - all we need to do is to open a file stream to the one XML source file.
- Parser** - parser is overkill for this project. All we need is **XmlParts** and **Tokenizer** and a stack to push element pointers on and pop them off as we build the AST.