CSE687 Midterm #4

Name: _____ SUID: _____

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 Exams 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 the easiest.

//

A good answer to a question about ideas and principles discusses the ideas and principles, how they will be used, and gives brief examples.

A good answer to a code question is brief, clear code that does what is asked and very little else. You can discuss elaborations of the code in comments.

I note the irony that I sometimes elaborate functionality of my code due to an over-abundance of enthusiasm, although that is usually done to instruct.

Grades are based on both contents of answer and clarity of presentation, but not on correctness of English grammar.

//

1. Write all the code necessary to find all the parents of a DbElement<P>, stored in a record of the NoSqlDb you implemented for Project #1. You may add code to the DbCore<P> if you wish. Remember that each database record's metadata contains a collection of children, not parents, of the file represented by the record.

```
//----< find all parents of record indexed by key >-----------------

template<typename P>
bool DbElement<P>::containsChildKey(const Key& key)  // you probably already defined
{                                                    // this DbElement method
  Keys::iterator start = children_.begin();
  Keys::iterator end = children_.end();
  return std::find(start, end, key) != end;
}

template<typename P>
Parents DbCore<P>::parents(const Key& key)
{
  Parents parents;  // alias for std::vector<Key>
  for (auto item : dbStore_)
  {
    if (item.second.containsChildKey(key))  // must be parent of key
      parents.push_back(item.first);
  }
  return parents;
}

// using code:

testKey = "Prashar";
Utilities::title("finding parents of " + testKey);
Parents parents = db_.parents(testKey);
showKeys(parents);
```

Note:
Notice how easy it is to understand what these functions are doing. That's craft.

It might be useful to define a query like this, so you don't have to search all the keys in the db, just the keys returned from a prior query.

2. Name[1] all the design principles discussed in class. Which of these does the XmlDocument facility you've used in Project #2, satisfy, and which does it not satisfy?

   Liskov Substitution is used because each of the concrete XmlElement types are used polymorphically, via base class pointers, e.g., std::shared_ptr<AbstractXmlElement>[2]. Each of the concrete types is accessed through the interface AbstractXmlElement and created using factory functions, makeTaggedElement, makeTextElement, etc., so the **Dependency Inversion Principle** is also used here.

   The **Interface Segregation Principle** is not used for the XmlElement classes, as all access to XmlElements occurs through the same interface. However, it is used because the XmlDocument class uses an interface specific for handling document activities, while the XmlElement classes use the AbstractXmlElement interface for handling all element configuration.

   The **Open/Closed Principle** could be used to add new XmlElements, but that is unlikely to be needed. XML has a standard structure that hasn't changed for years.

   Single Responsibility Principle is used effectively in the XmlElement hierarchy. Each derived class, XmlDeclarElement, DocElement, TaggedElement, TextElement, CommentElement, and ProcInstrElement, focus on configuring a specific XML element type. The XmlDocument class focuses on managing the XML document model.

   Least Knowledge Principle is used, by hiding the parsing mechanics in classes XmlElementParts and XmlParser, so users and XmlElement classes don't have to be aware of the parsing mechanics.

   Value Semantics Principle is not used, as the XmlElement classes and XmlDocument class remove copy operations with the =delete syntax.

   Scope-based Resource Management Principle is used because all XmlElement instances are accessed through std::shared-ptr<AbstractXmlElement> smart pointers, which do scope-based, reference counted, resource management.

   The Encapsulation Principle is used in the XmlDocument and XmlElement class hierarchy, as they keep all of their data private.

   Note: All principles discussed here are documented in CSE687 > Notes > Design Principles. The first four items are a complete answer to this question, for grading, as those are the ones I've stressed in class. You need a small amount of discussion to make convincing arguments about whether these prinicples are satisfied or not.

---

[1] This question is asking you to name, not define or state the principles.

[2] Technically, the base class pointer, AbstractXmlElement*, is a data member of the std::shared_ptr<AbstractXmlElement> class, but the standard smart pointers all work polymorphically, in the same way that their data member pointers, work.

3. Describe what happens when the last statement below is executed:

```
std::unordered_map<std::string, double> aMap;
--- some code that uses aMap ---
aMap[“Zhang”] = 23;
```

The literal string “Zhang” is promoted to the type of the map’s key since there is a suitable promotion constructor.

This key is used in a hash function to find a specific bucket, e.g., a linked list of key-value pairs with root stored at the address computed by the hash function.

If the key “Zhang” exists in the unordered_map’s bucket, then it’s value is overwritten with the value 23. If that key does not exist in the unordered_map, then a new std::pair:

std::pair<std::string, double> = { “zhang”, 23 }

Is inserted into the unordered_map, using the map’s hash function for the key to find the bucket where the std::pair will be stored.

Note:
no std::unordered_map assignment is called. To answer this question fully, you have to explain what happens if the key exists and also what happens if the key does not exist.

See MTS18-code.MT4Q3 for a demo.

4. Write all the code for a Node class that holds a variable of unspecified type and can link to other child nodes. Instances of this class could be used by a Graph class to represent its vertices[3].

```
// Pseudo code:
// Node<T> is container class that holds
//    T value
//    std::vector<std::shared_ptr<Node<T>>> children
// and provides methods to add and access them
```

```
template<typename N>
using Sptr = std::shared_ptr<N>;
using Key = std::string;

template<typename V>
class Node
{
public:
  Node(const std::string& name) : name_(name) { }
  V value() const { return value_; }
  void value(V v) { value_ = v; }
  void addChild(Sptr<Node<V>> pNode) { children_.push_back(pNode); }
  std::vector<Sptr<Node<V>>>& children() { return children_; }
  // not needed for answer ///////////////////////
  std::string& name() { return name_; }
  void mark() { visited_ = true; }
  void unmark() { visited_ = false; }
  bool& marked() { return visited_; }
  ////////////////////////////////////////////////
private:
  V value_;
  std::vector<Sptr<Node<V>>> children_;
  // if you hold vector<Node<V>> you will have redundant nodes in graph
  std::string name_;  // not needed for answer
  bool visited_;      // not needed for answer
};
```

This class assumes that V is copy constructable and copy assignable. Since std::shared_ptr, std::vector, and std::string are all copy constructable and copy assignable, the class should allow the compiler to generate copy operations and destructor.

Note:
I've changed the syntax of the template definitions from the original in GraphWalkDemo. I think this version is a little clearer, although a bit more verbose. I believe both are correct.

I didn't expect you to remember all this. I did expect you to be able to recreate something close, based on the operations needed for a Node class.

Use of std::shared_ptr is essential here. Otherwise, there is no way for the class to manage child nodes. Are they on heap? In static memory? Does the user intend to use them elsewhere? The std::shared_ptr<Node<V>> takes care of all these problems. It is a reference counted manager of resources always stored on the heap.

A fully correct answer could omit the marking operations and the handling of node names.

[3] You don't have to write any code for the graph class.

5. Describe how you would safely share a string between two C++11 threads where either of the threads may read or modify the string. Please be specific.

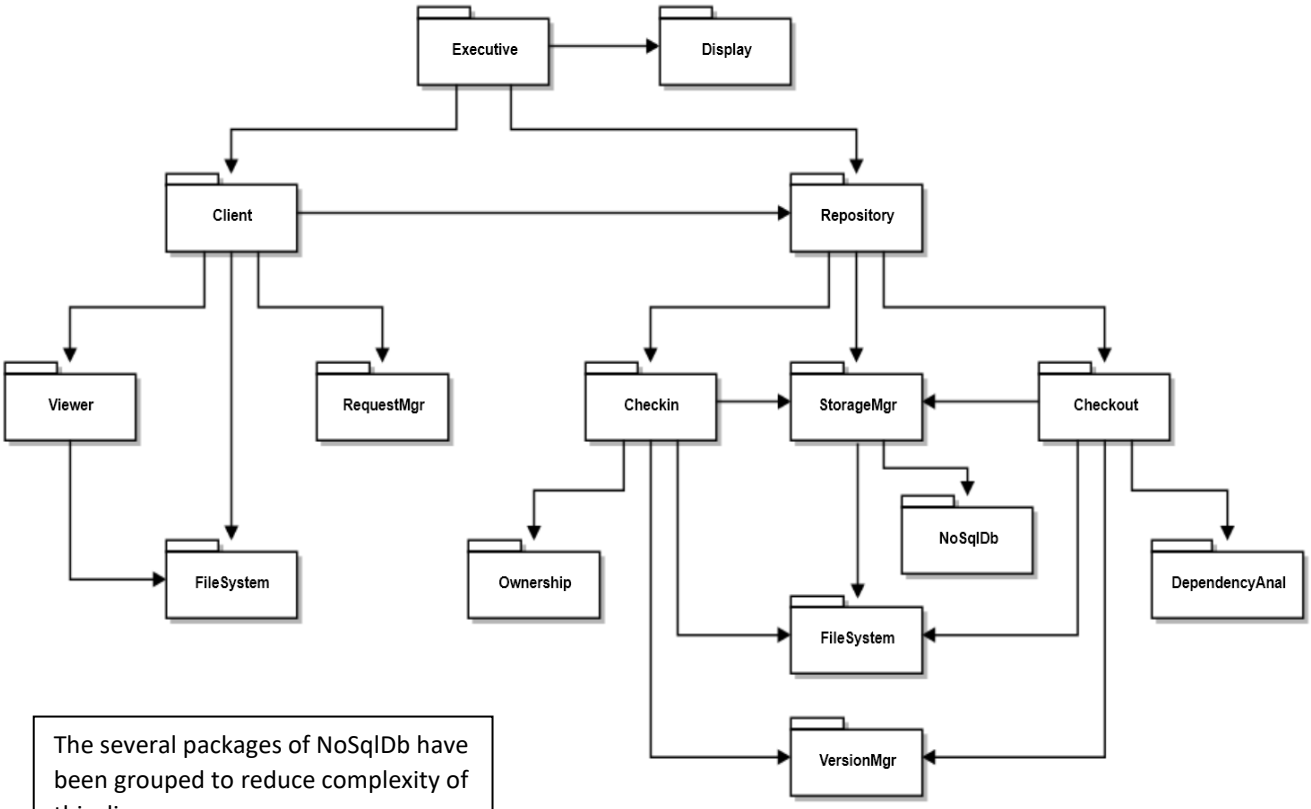
You could:
Create a thread function that accepts a string by reference and modifies it.
In the body of the function declare a static std::mutex and use it directly, or wrapped by std::unique_lock, to enforce exclusive access for a calling thread.
Declare threads to share the string, passing the shared string by std::ref.
Ensure that the client does not access the string until sharing is complete, perhaps by joining.

```
//----< concurrent sharing happens here >------------------------------

void modifyString(const std::string& name, std::string& shared)
{
  static std::mutex mtx;
  std::unique_lock<std::mutex> lck(mtx, std::defer_lock);

  for (size_t i = 0; i < 5; ++i)
  {
    lck.lock();
    shared += name;
    lck.unlock();
    std::this_thread::sleep_for(std::chrono::milliseconds(20));
  }
}
//----< protect shared string from outside access >--------------------

std::string makeCopyOfSharedString(std::string shared)
{
  std::thread t1(modifyString, "Gabe ", std::ref(shared));
  std::thread t2(modifyString, "Ankur ", std::ref(shared));
  std::thread t3(modifyString, "Tianyu ", std::ref(shared));
  // client can't use shared string until threads are done
  // so we might as well join, but even so, sharing is
  // executed in parallel
  t1.join();
  t2.join();
  t3.join();
  return shared;
}
```

To allow client to do other things while this runs, simply do this:
`std::future<std::string> result = asynch(makeCopyOfSharedString, shared);`
Since asynch arguments are passed by value you can rename the function and pass by reference there to avoid two copies.

Note:
Accessing a global variable using threads is dangerous, even if all the thread procedures lock the variable with a shared lock. There is no way to ensure that some other function isn't also reading or writing the global variable. Unfortunately, in one of my demos I made a shared string publically accessible, so in fairness, I didn't take off points for that.

6. Draw a package diagram for your design of the Repository for Project #2.



Note:
If you used class relationship symbols on your package diagram you lost significant score.
You received some bonus if you described the package responsibilities even though not required.

7. When will a standards-conforming C++11 compiler invoke a move operation on an instance of a class that provides that operation?

   Only when the move operation (construction or assignment) is defined by the class, or by the compiler for the class. Then, only when:

   a. The instance is passed by value from a temporary or assigned from a temporary.
   b. That also happens for non-temporaries if we use the std::move(theInstance). That may invalidate theInstance, so it should not be used after the move.
   c. That may also happen if we use std::forward(theInstance)[4]. We will discuss that syntax when we talk about template metaprogramming.

   Here are some obvious cases:

```
s = s1 + s2;          // all strings, rhs is an unnamed temporary
V v = someFun();      // someFun returns V instance created internally
v1 = std::move(v2);   // v2 is lvalue, cast to rvalue type by move(v2)
```

   Note:
   If a class makes its copy constructor and assignment operator =delete, that does not mean that return by value or assignment will be moves. If the class does not define a move operation and the compiler can't generate one, then the statement implying copy or assignment will fail to compile.

[4] You were not expected to know about std::forward, since we haven't discussed it yet.