CSE687 Midterm #3

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. Quality of English grammar was not considered.

///

1. Write all the code for a class that wraps any C++ primitive[1] to enable instances of the class to behave like the primitive[2], but, may add additional functionality. Hint: you will need constructors and assignment operator to accept a primitive value. You will also need a cast operator to retrieve the primitive value. Can you think of a design situation where that might be useful?

Box class is the complete answer for "write all code …"

```
 ///////////////////////////////////////////////////////////////////////
 // Box class – see MTCode-S18.CodeUtilities
 // - wraps primitive type in class, so it can be a base for inheritance
 // - preserves primitive syntax

 template<typename T>
 class Box
 {
 public:
   Box() : primitive_(T()) {}
   Box(const T& t) : primitive_(t) {}
   operator T&() { return primitive_; }  // cast operator
   T& operator=(const T& t) { primitive_ = t; return primitive_; }
   virtual ~Box() {}
 private:
   T primitive_;
 };
 ///////////////////////////////////////////////////////////////////////
 // ToXml interface - defines language for creating XML elements

 struct ToXml
 {
   virtual std::string toXml(const std::string& tag) = 0;
   virtual ~ToXml() {};
 };
 ///////////////////////////////////////////////////////////////////////
 // PersistFactory<T> class
 // - wraps an instance of user-defined type, preserving semantics of user-defined type
 // - adds toXml("tag") method
 // – this is a "useful" design situation.  See MTS18-Code.Utilities for details
 template<typename T>
 class PersistFactory : public T, ToXml  // T can't be primitive, but
 {                                       // can be Boxed primitive
 public:
   PersistFactory() = default;
   PersistFactory(const T& t)
   {
     T::operator=(t);
   }
   std::string toXml(const std::string& tag)
   {
     std::ostringstream out;
     out << "<" << tag << ">" << *this << "</" << tag << ">";
     return out.str();
   }
 };
}
```

[1] A primitive type is a type defined by the C++ Language. Examples are int and double.
[2] This is similar to boxing in C#.

2. State the interface segregation principle (ISP). Have you applied ISP in your implementation of Project #1?

   "Clients should not be forced to depend upon interfaces they do not use".

   We have used ISP extensively in Project #1. The facilities for making queries and persisting the NoSqlDb to XML have been segregated from the core db functionality. A client uses them only if they need to make queries using our facilities or to persist the db. Other instances are the use of Edit and TestClass packages. This is clearly illustrated by my answer to MT2-Q2.

   Defining one interface, like IPayLoad, and changing it for different applications, is not ISP.

   Note that following the Single Responsibility Principle is very likely to segregate interfaces, as described above. However, following Interface Segregation Principle does not imply that SRP is also used, so these two principles are not equivalent.

   .

3. Write all the code for a function that accepts a callable object and executes it. If execution throws an exception, the thread running your method should sleep for a specified amount of time, and then retry. It should do this for a specified number of times before returning a failure to execute.

There is no requirement to use Ret, Arg, or ErrorPolicy. See MT3Q3b code.

```
//----< retry execution >--------------------------------------------//
/*
   - invokes callable object co
   - if exception is thrown, retries maxCount times
   - sleeps duration milliseconds between each try
   - tricky part is returning error value, which we do with ErrorPolicy
   - this is a great demonstration of why we may need template policies
   - uses callable signature Ret Callable(Arg), useful for opening streams.
     Ret is the opened stream, arg is the fileSpec to open.
*/
template <typename Ret, typename Callable, typename Arg, typename ErrorPolicy>
Ret reTry(
  Callable co,
  Arg arg,
  ErrorPolicy ep,
  std::chrono::duration<size_t, std::milli> duration = std::chrono::milliseconds(100),
  size_t maxCount = 10
)
{
  size_t count = 0;
  while (true)
  {
    try
    {
      Ret ret = co(arg);
      std::cout << "\n  operation successful";
      return ret;
    }
    catch (...)
    {
      if (++count < maxCount)
      {
        std::cout << "\n  operation failed - retrying";
        std::this_thread::sleep_for(duration);
        continue;
      }
    }
    std::cout << "\n  operation failed";
    Ret ret;                  // didn’t get callable’s return so create a default
    ep.setErrorState(ret);  // set ret’s error state using template policy

    ////////////////////////////////////////////////////////
    // This is what setErrorState does for streams:
    //   std::ios* pStream = dynamic_cast<std::ios*>(&ret);
    //   if (pStream != nullptr)
    //     pStream->setstate(std::ios::failbit);
    ////////////////////////////////////////////////////////

    return ret;
  }
}
```

4. You are preparing to design a class that may need to be extended in the future, in ways you can't anticipate now. How might you do that?

   Here are four common ways, and one not so common way, to make our code extensible:

   a. Use templates with template policies. We can modify the way a template class behaves by substituting appropriate policy classes for different application needs. See MT3Q3, above.
   b. Provide template method(s) that accept callable objects. See the Query class in MTS18-code for an example.
   c. Use inheritance and polymorphic classes. We can extend an inheritance hierarchy simply by adding new derived classes that override base virtual functions. The parser is an outstanding example of the flexibility afforded by the use of inheritance.
   d. Use interfaces and object factories. This is an elaboration of the item, c., above. If we implement an interface with code that compiles to a dynamic link library, then we can extend the existing functionality by adding a new library that implements the interface and modifying the small amount of code in the object factory to create instances needed in the new library. See XmlElement for example.
   e. Add additional interfaces and support querying from one interface for another supported interface, as illustrated in the Object Factory demo in Lecture #12.

5. What methods are not inherited by a derived class from its base class. How does the C++ language ensure that Liskov Substitution works for derived class pointers and references in code that may use those methods.

   The constructors (copy and move), assignment operators (copy and move), and destructors **used** in the derived class are not inherited from the base by derived classes[3]. However, the compiler will generate those operations, as follows:

   a. A default constructor will be generated only if no constructors are declared.
      Copy constructors will always be generated, if needed and not declared.
   b. Move constructors will only be generated if no copy operations, assignment operations, and no destructor are defined.
   c. Copy assignment will always be generated, if needed and not declared.
   d. Move assignment will only be generated if no copy operations, assignment operations, and no destructor are defined.
   e. A destructor will always be generated, if not declared.

   The compiler generated operations simply apply the operation to each of their bases and data members. These will be correct if each of the bases and members have correct construction, assignment, and destruction semantics[4].

   Since **all other** methods used by the derived class are inherited[5], any call made using a base reference or pointer can be made on a derived reference or pointer. That would not be the case if the compiler did not generate those methods.

   Note that "**all other**", above includes non-virtual and private methods of the base. Yup, they are inherited. We demonstrated that with the Non-Virtual Interface Principle demo in Lecture #10.

[3] Technically **ALL** methods are inherited, but constructors, assignment operators, and destructor, **USED** by the derived class are not inherited. The inherited assignment is always hidden by explicitly defined or compiler generated assignment operators, but they can, and often do, explicitly invoke the base operations. All other operators are inherited as well. Similarly, the derived constructors often explicitly invoke the inherited base constructors to implement their operations.

[4] In is incorrect to say that the compiler generated functions are shallow. They simply delegate to the base and member operations, which, if needed and correct, are deep.

[5] So **EVERY** method of the base class is present in the derived class, either by inheritance or by compiler generation. That, of course, can be broken by using =delete in the derived class and not in the base for compiler generated methods.

6. What are template policies and template traits? Have you used them in your implementation of Project #1?

   A template policy is a template parameter[6] that is used to modify the operation of a template class, e.g., it does not replace the primary functioning of instances of the class, but only modifies how they carry out their operations. See the answer to question 3 for an example. You could also argue that the parameter of the template method in Query class used to accept callable objects is a policy, e.g., a policy that modifies Query instances to support user-defined query operations.

   A trait is an alias for a template parameter type that provides an immutable name for that type, regardless of how it is instantiated by an application. Traits allow us to write code in a template function or class that depends on the specific type of a template parameter without knowing how the application will instantiate it. The DbCore<P>::iterator, from Project #1, is an example.

[6] A template policy is almost never the primary template parameter. I would not classify PayLoad as a template policy in the NoSqlDb classes.

7. How can you safely retrieve a value computed by a C++11 thread?

There are two common ways to do that. First, and preferred, is to use std::async which returns a std::future with the result. Second, you can pass a C++ std::thread an argument by reference, then, after joining, access the value of the argument, assuming the thread's function stores its result in the reference argument.

No code is required for this question.

```
size_t theFirstMeaningOfLife()
{
  std::cout << "\n  The meaning of life: ";
  for (size_t i = 0; i < 5; ++i)
  {
    std::this_thread::sleep_for(std::chrono::milliseconds(500));
    std::cout << "thinking ... ";
  }
  return 42;
}

void theSecondMeaningOfLife(size_t& answer)
{
  std::cout << "\n  The meaning of life: ";
  for (size_t i = 0; i < 5; ++i)
  {
    std::this_thread::sleep_for(std::chrono::milliseconds(300));
    std::cout << "thinking ... ";
  }
  answer = 42;
}

int main()
{
  std::cout << "\n  MT3Q7 - Getting a std::thread result";
  std::cout << "\n ======================================\n";

  // async with future is thread safe by design
  std::future<size_t> fut = std::async(theFirstMeaningOfLife);
  std::cout << "\n  the first meaning of life is: " << fut.get();
  std::cout << "\n";

  // for thread safety, must join before accessing ref value
  size_t mol;
  std::thread t(theSecondMeaningOfLife, std::ref(mol));
  t.join();
  std::cout << "\n  the second meaning of life is: " << mol;

  // prefer the first method
  std::cout << "\n\n";
  return 0;
}
```

You could also use a callback to do something application specific with the result, but that isn't directly retrieving the computed result, as the child thread runs the callback .