CSE687 Midterm #2

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.

//

1. State the Open/Closed principle and describe how you plan to use that in your implementation of Project #2.

   Software entities (classes, packages, functions) should be open for extension, but closed for modification.

   Software Repositories have a set of expected functionalities: authorization, storage management, versioning, check-in, check-out, dependency analysis, generation and recording of metadata, browsing, and request management. Using organizations may have differing requirements of almost all of these, e.g.: What is the ownership policy? How will stored items be placed in folders? How are versions structured, recorded, and associated with individual repository items? What are the rules for check-in (when will that succeed, when fail, when are modifications allowed)? What are the rules for check-out and what is actually returned? What metadata should be recorded for each item? How will users be allowed to traverse items in the Repository? What interface will be provided for client actions and queries?

   We will satisfy the Open/Closed principle if we support substitution of Repository modules to accommodate these different usage styles. We do that by providing interfaces and object factories for Ownership, Storage, Version, Check-in, Check-out, and Browsing. This allows users to "plug-in" modules that suit their specific needs without changing any of the other components, e.g., modify by extension, not by altering Repository code. Essentially, we satisfy OCP by using DIP.

   Instead of using inheritance, one could use templates to segregate the parts that change with each application from those that should be invariant. In particular, template policies for Ownership, Storage, Check-in, Check-out, etc. would be an effective use of OCP for Project #2. Using template members that accept callable objects is also very effect. You will see that in my Query class in the NoSqlDb prototype.

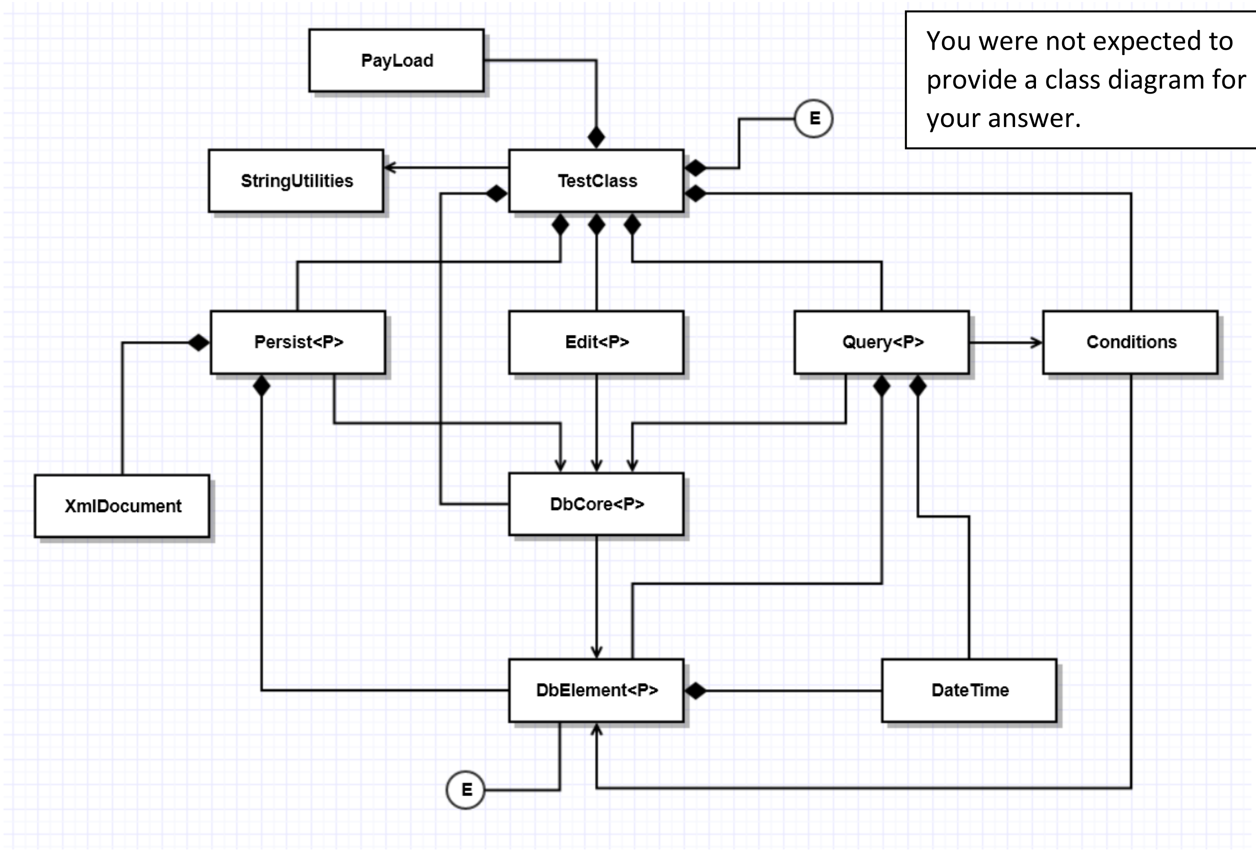
   See MT1Q4 solution for an example of how to use templates to separate application code from reusable code. It can sometimes be effective to use both template policies and inheritance in the same class, for extension.

   Note, also, that the operation of the Repository itself supports use of the Open/Closed principle by providing access to reusable components to build future systems, especially by making each version, when closed, immutable.

   Note: This is one of many plausible answers for this question. You have been given a lot of latitude in the specifics of how you answer the question, provided that you have been **specific**. However, it is intended that you discuss how **YOU** will support OCP in **Project #2**, not how class demo code did that, or how it could be done for arbitrary code.

2. Using the design principles and techniques discussed in class this semester, evaluate[1] your design of Project #1.

You were not expected to provide a class diagram for your answer.



My solution for Project #1, the NoSqlDb, illustrated by the class diagram, above, was factored into a number of classes, Persist, Edit, and Query, that use the DbCore class to provide functionality for Project #2. DbCore is a reusable component that has been extended to support Persistance, Querying, and Editing of records, without modifying the Core db in any way. These classes, themselves, will be extended by instantiating instances with a PayLoad class to manage application specific data. The **Open/Closed Principle** is very effectively satisfied by this design. Furthermore, each of the classes shown above satisfies the **Single Responsibility Principle**. They each focus on performing one activity effectively.

The **Dependency Inversion Principle** was used only in the XmlDocument and especially XmlElement class hierarchy. Each of the XmlElement classes derives from an AbstractXmlElement, which provides an interface for all of the concrete element classes. Also, each of the element classes has a factory function, responsible for creating and initializing instances of the elements. These classes are used polymorphically, satisfying the **Liskov Substitution Principle**.

Since each of the classes DbCore, Persist, Query, and Edit provide their own languages for interacting with db data, this design also satisfies the **Interface Segregation Principle**.

[1] Evaluate means, for all the major parts of your design, you state whether it satisfies the principle, and very briefly how it does that, e.g., no more than one sentence.

3. Assume you have designed, for Project #1, a Query class that provides a template method accepting callable objects. Write all the code for a lambda that is used to query your database, implemented in Project #1, for all the elements that have a specified child, when passed to the template method. Write one or two statements to show how you would use the lambda.

```
template<typename P>
class Query
{
public:
  Query(DbCore<P>& db) : db_(db) { keys_ = db_.keys(); }
  static void identify(std::ostream& out = std::cout);
  Query& select(Conditions<P>& conds);

  template<typename CallObj>       // Applies CallObj to each element with
  Query& select(CallObj callObj);  // key in query's orig. key collection. Puts key
                                   // in new key collection if CallObj returns true.
  Query& query_or(Query<P>& q);
  Query& from(const Keys& keys) { keys_ = keys; return *this; }
  void show(std::ostream& out = std::cout);
  Keys& keys() { return keys_; }
private:
  DbCore<P>* pDb_;  // First version used reference DbCore<P>& db_.
  Keys keys_;       // Changed to pointer to support changing db.
};

template<typename P>
template<typename CallObj>
Query<P>& Query<P>::select(CallObj callObj)
{
  Keys newKeys;
  for (auto key : keys_)  // iterating over query keys
  {
    if (callObj((*pDb_)[key]))
      newKeys.push_back(key);
  }
  keys_ = newKeys;  // replace original keyset with newKeys
  return *this;
}
```

//
// answer starts here:

```
std::cout << "\n  select on child key \"Arora\"\n";
Key aChild = "Arora";
auto findChild = [&aChild](DbElement<PayLoad>& elem) {
  for (auto child : elem.children())
  {
    if (child == aChild)
      return true;
  }
  return false;
};

q1.from(db_.keys()).select(findChild).show();  // from argument could be
                                               // keyset from prior query
```

4. Write all the code for a function that searches for a string of text in a text file, asynchronously[2]. Assume that you pass the file name as a function argument.

```
using FPtr = bool(*)(const std::string&, const std::string&);

bool findText(const std::string& text, const std::string& filePath)
{
  std::ifstream in(filePath);
  if (!in.good())
  {
    std::cout << "\n  can't open \"" << filePath << "\"";
    return false;
  }
  std::string fileText;
  while (in.good())
  {
    char ch;
    in >> ch;
    if(ch != ‘\n’)  // text may span multiple lines
      fileText += ch;
  }
  size_t pos = fileText.find(text);
  return pos < fileText.size();
}

std::string text = "main";

std::string fPath1 = "../Test/Copy_MT2Q4.h";
std::string fPath2 = "../Test/Copy_MT2Q4.cpp";

std::future<bool> f1 = std::async(std::launch::async, findText, text, fPath1);
std::future<bool> f2 = std::async(std::launch::async, findText, text, fPath2);
```

This while loop can be replaced by:

```
  Std::ostringstream out;
  Out << in.rdbuf();
  Std::string fileText = out.str();
```

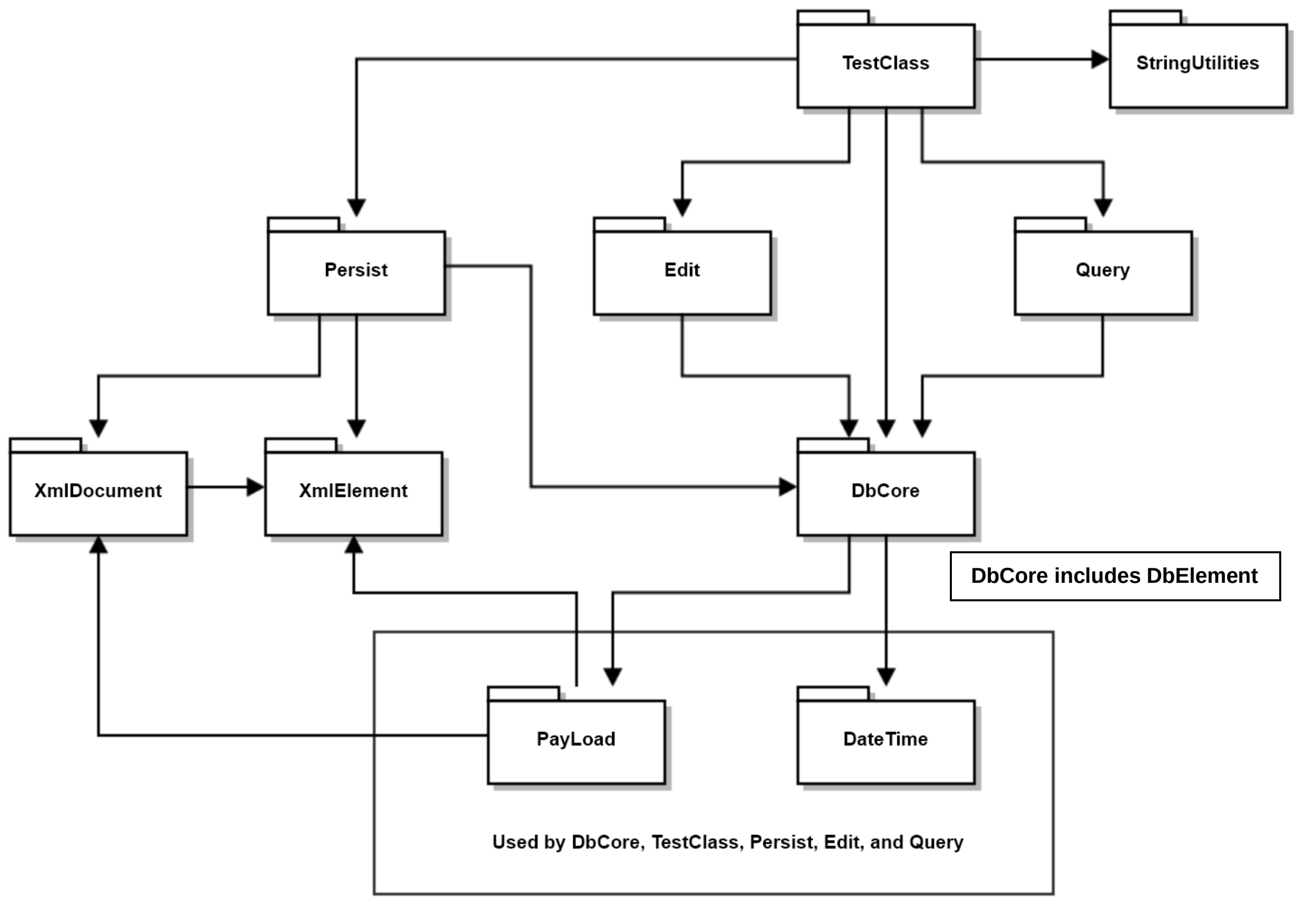
We will discuss this when we talk about iostreams.

[2] When you run the function asynchronously, the caller returns quickly, perhaps before the passed lambda finishes execution.

5. Declare an interface for ownership policies for Project #2. It is up to you to decide what methods the interface should have. Why might you choose to use an interface for ownership?

```
C++ interfaces have all public pure virtual methods, except for a virtual
destructor with empty body.  They have no data members and no
constructors.

namespace Repository {

    using Owner = std::string;
    using Owners = std::vector<std::string>;
    using Item = std::string;  // namespace::filename.ext.ver

    struct IOwnership  // all members public by default for structs
    {
        bool authenticate(const Owner& name, const Item& item) = 0;
        Owners owners() = 0;
        bool addOwner(const Owner& name) = 0;
        virtual ~IOwnership() {}
    }
}


See answer to MT2Q1 for why you might choose to use this interface.
```

6. Draw a package diagram for your implementation of Project #1.



7. What class methods may be generated by a standard conforming C++ compiler? When would those operations be generated?

   The following operations are, if needed and not declared, provided by the compiler. Each delegates its operation to each of the class's bases and data members.

   a. Default constructor:
      Provided only if no constructors are declared for the class.
   b. Copy constructor:
      Always provided if not declared by the class and used in application code.
   c. Move constructor:
      Only provided if not declared by the class and copy and destructor operations are not declared by the class.
   d. Copy assignment:
      Always provided if not declared by the class and used in application code.
   e. Move assignment:
      Only provided if not declared by the class and copy and destructor operations are not declared by the class.
   f. Destruction:
      Always provided if not declared by the class.

Note:
Classes, that have bases and data members with correct construction, assignment, and destruction semantics, should not declare these methods. Classes that use only primitive data and STL containers have correct semantics for these operations, and you should not provide them.

Classes that hold owning pointers should almost always provide the copy, assignment, and destruction operations. Move operations are optional, but, may make significant improvements in performance.

Template methods are completed with instantiated type information, but they are not generated by the compiler.