CSE687 Midterm Spring 2018

Questions and Answers

These questions, with answers, were taken from my solutions for the four midterm exams given in the Spring 2018 Semester, for CSE687 – Object Oriented Design.

//

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.

//

Questions are ordered by topic, with this order:

Questions about Object Oriented Design

"OOD is the design of software using classes and class relationships."

These questions include Design Principles and methods for building flexible code that is easy to change and test.

You cannot learn OOD, C++, Software Design, and System Architecture, without writing a lot of code. These topics are big and complex, so you have to:

```
While(true)
    Read a topic
    Write code relevant to that topic
    Choose another topic
End While
```

1. State the Liskov Substitution Principle and describe ways to write code that can cause substitution errors.

   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.

   Code constructs that may cause substitution errors:
   a. Overloading base class methods in a class derived from the base – causes hiding
   b. Overloading base class virtual functions – causes hiding when those functions are overridden in the derived class.
   c. Redefining in a derived class a non-virtual method in the base – non-virtual method calls are based on the type of the pointer, not the type of the object bound to the pointer.
   d. Failure to provide a virtual destructor in a class that will serve as a base for inheritance – causes incomplete destruction of derived objects on the heap when deleted from a base class pointer bound to the derived instance.
   e. Use of default parameters in both base and derived with different values if accessed through a base pointer or reference – will use values determined by the static type of the pointer or reference, not the type of the bound instance.

2. What is object oriented design (OOD)? How have you used OOD in your first project? Please be specific.

   OOD is the design of software using classes and class relationships.

   The Project #1 NoSqlDb is composed of many classes with single responsibilities. The list, below, describes each class and shows their relationships with other classes:
   a. DbCore – record management
   b. PayLoad – application specific data management
   c. Edit – modify existing db records
      uses instance of DbCore
   d. Query – extract information from db
      uses instance of DbCore
   e. Persist – Save and retrieve db records
      owns XmlDocument, uses instance of DbCore
   f. XmlDocument – support creating XML representation of db and recreation of db records
   g. DateTime – manage time information
   h. Utiltites – support for type conversions and display
   i. Executive – compose all Project #1 functionality and deploy
      owns instances of DbCore, PayLoad, Edit, Query, and Persist
      uses Utilities
   j. TestClass – demonstrate satisfaction of requirements
      owns instances of DbCore, PayLoad, Edit, Query, and Persist
      uses Utilities

   Several of you have said that OOD is the use of Object Oriented Languages. That is true for Java and C#, but is incomplete. One can use C++ without using any classes even though it is an excellent Object Oriented Language.

   Some have cited using templates, but that is not OOD. It is generic programming. That, of course, does not mean that a template class is not an artifact of OOD, but rather, the template has nothing to do with its OOD-ness.

   Several of you also discussed packages instead of classes in Project #1. Packages are not OOD specific. C language programs use packages, for example.

3. 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 effective. 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.

4. 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.

5. 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.

6. 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.

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

   A template policy is a template parameter[3] 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.

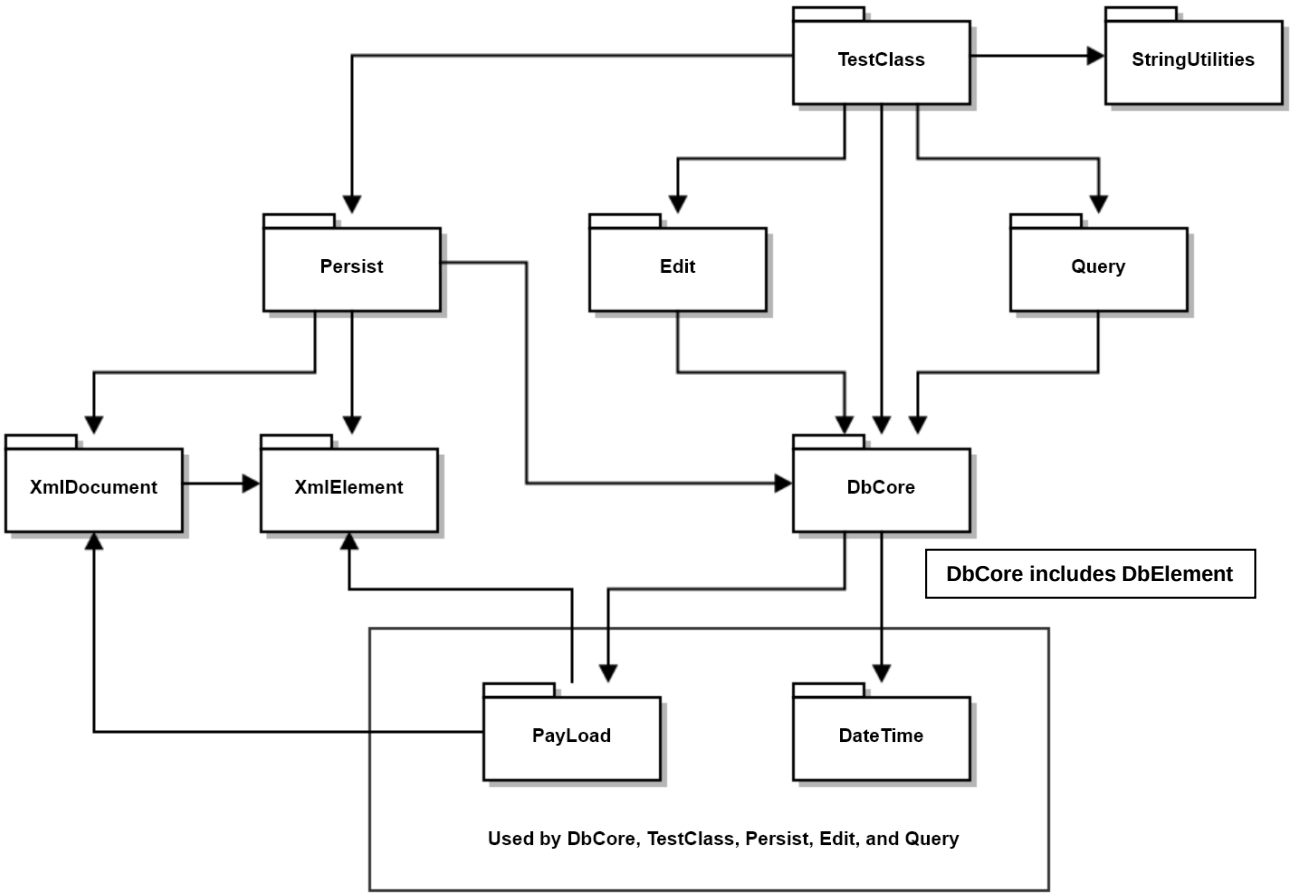
   A trait is an alias for a template parameter type that provides an immutable name for that type, regardless of how it is instantiated by an application. Traits allow us to write code in a template function or class that depends on the specific type of a template parameter without knowing how the application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

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

Project #1 Design

Project #1 builds a No SQL Database. This is an in-memory, key-value based container for arbitrary data described by a template parameter P, representing some data structure useful for a specific application. It also contains metadata that includes child dependency relationships, e.g., a collection of keys that identify db elements that are the dependency children of the db element containing the collection.

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



9. Assume you have designed, for Project #1, a Query class that provides a template method accepting callable objects. Write a query, using that method, that finds all files with a specified category as recorded in the payload instance[4] for that file.

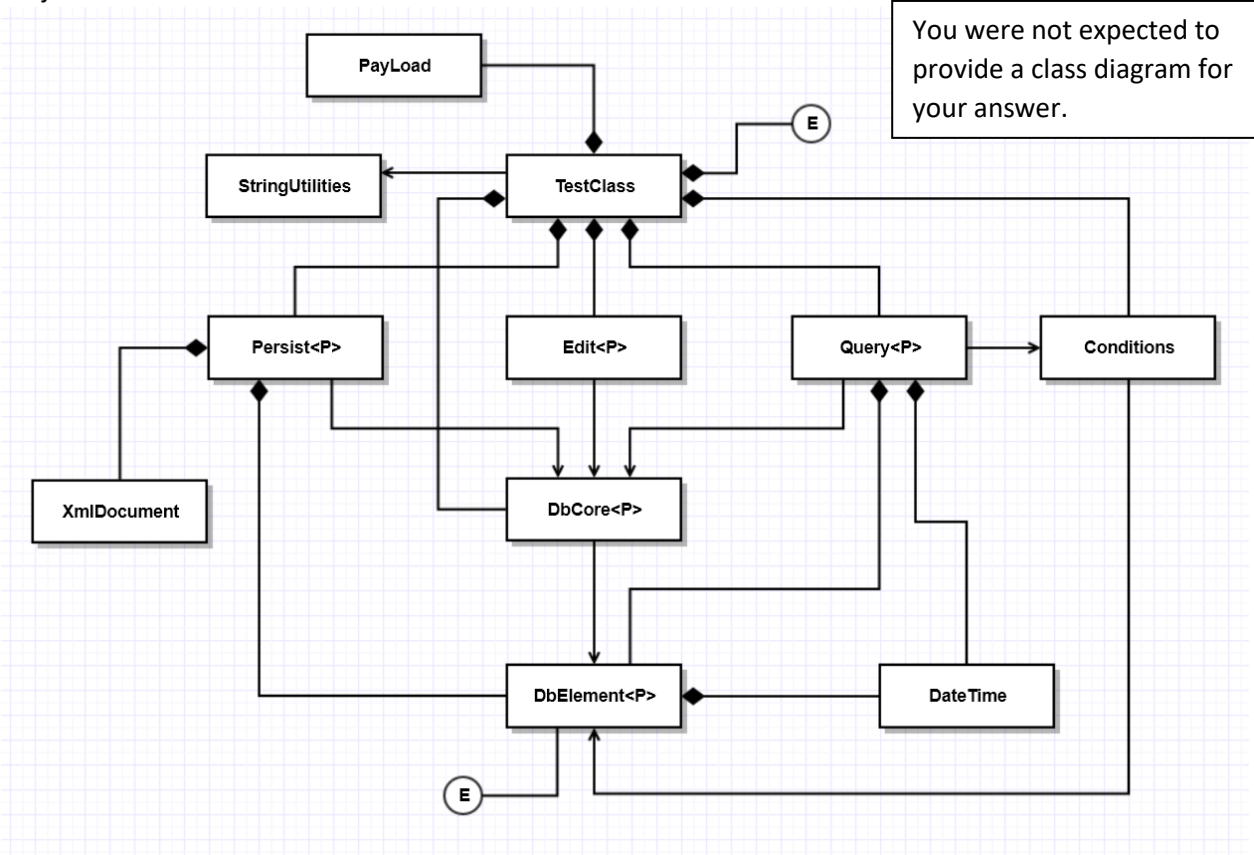
```
template<typename P>
class Query
{
public:
  Query(DbCore<P>& db) : db_(db) { keys_ = db_.keys(); }
  static void identify(std::ostream& out = std::cout);
  Query& select(Conditions<P>& conds);

  template<typename CallObj>
  Query& select(CallObj callObj);  // applies callObj to each DbElement in
                                   // query's key set
  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>& db_;
  Keys keys_;
};

template<typename P>
template<typename CallObj>
Query<P>& Query<P>::select(CallObj callObj)
{
  Keys newKeys;
  for (auto key : keys_)  // searches current keyset
  {
    if (callObj((*pDb_)[key]))
      newKeys.push_back(key);
  }
  keys_ = newKeys;       // return holding new keyset
  return *this;
}
/////////////////////////////////////////////////////////////////
// answer starts here:

Query<PayLoad> q1(db_);

std::string category = "thirdCategory";
std::cout << "\n  select on payload categories for \"" << category << "\"\n";

auto hasCategory = [&category](DbElement<PayLoad>& elem) {
  return (elem.payLoad()).hasCategory(category);
};

q1.select(hasCategory).show();  // using template method
```

[4] You may assume that the PayLoad class has a method:

bool PayLoad::hasCategory(const std::string& category)

10. Using the design principles and techniques discussed in class this semester, evaluate[5] 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**.

[5] 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.

11. 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
                                               // key set from prior query
```

12. 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.

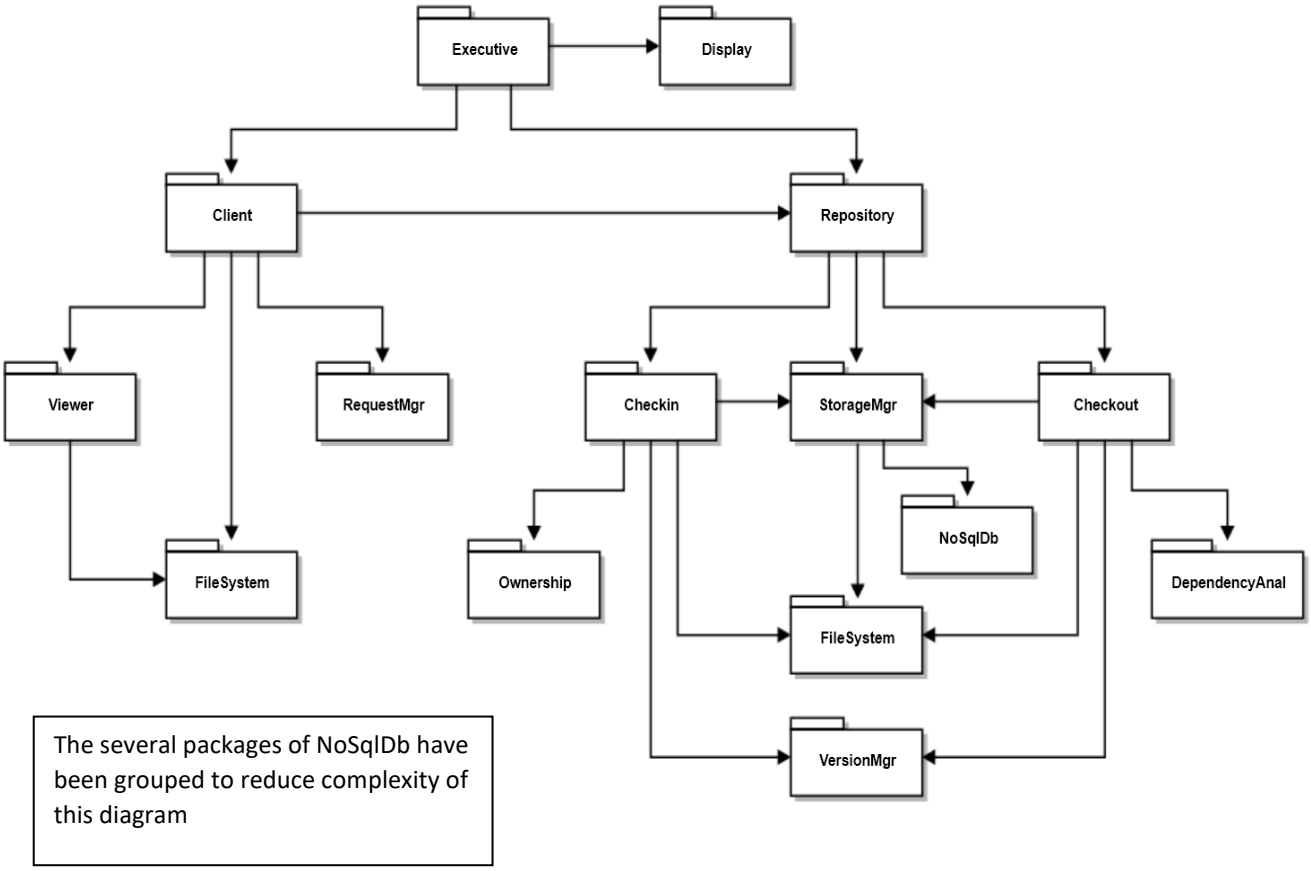
It might be useful to define a query like this, so you don't have to search all the keys in the db, just the keys returned from a prior query.

Project #2 Design

In Project #2, we build a Software Repository that is intended to store the current working baseline for a software development project, e.g., all of the files and documents that constitute what the developers believe to be the current state of project's product.

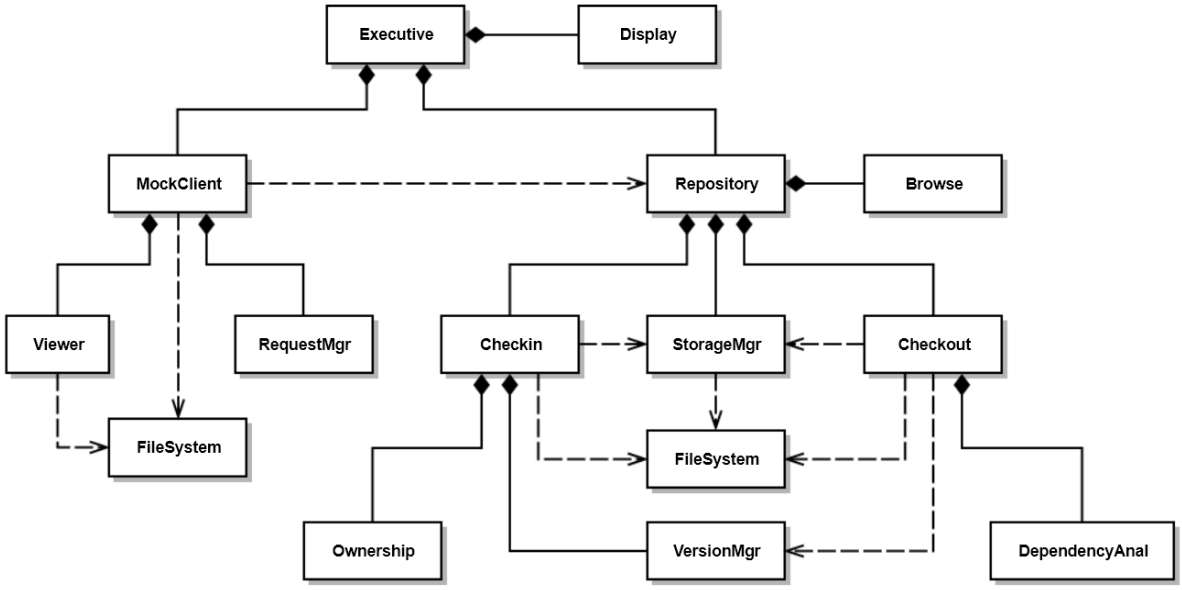
The Repository is expected to provide Check-in, Check-out, Versioning, Browsing, and Ownership services. It will store files and documents in directories, and store information about the current state of the repository in the NoSqlDb we built in Project #1, e.g., path to, and metadata about each file in the Repository storage.

13. 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.

14. Draw a class diagram for your design of Project #2. Discuss, briefly, the responsibilities of each. You do not have to include the NoSqlDb classes.



Class	Responsibility
Executive	Create MockClient and Repository and start execution
Display	Writes demonstration messages to console
MockClient	Initiate Checkin and Checkout operations
Viewer	Write request traffic to console (will become view in GUI – Pr#3), uses process to display file text. Diagram assumes process part of Viewer.
Repository	Create Checkin, Checkout, and StorageMgr Respond to MockClient requests
Checkin	Respond to checkin request, using Ownership rules and VersionMgr
Checkout	Respond to checkout request using DependencyAnal, VersionMgr
StorageMgr	Manages Repository Folders and placement of files
Browse	Provides name, description, and children of specified file
VersionMgr	Creates new versions and finds latest versions
Ownership	Validate checkin, may track check-outs for multiple owner policies
DependencyAnal	Evaluate transitive dependency relationships
FileSystem with classes: File, Path, and Directory	Provide directory and file management utilities. This is a stand-in for the classes File, Path, and Directory, to lessen complexity of diagram

This is a fairly elaborate design for Project #2. A simpler design may be acceptable.

Note that only two classes do not have owners: Executive and FileSystem which has only classes with static methods. Also note that the responsibilities describe actions, and in some cases, action flows.

15. Write a list of tasks that your check-in process must execute for Project #2. Please consider complete and incomplete inputs provided by the client.

Check-in tasks:

a. Validate check-in using ownership rules
    i. If invalid, fail checkin, notify client, and exit
b. Copy file from path specified by Client
c. Determine if this is a new check-in or continuation of an open check-in
    i. Check closed status in payload of last checkin if there is one
    ii. If closed or closed pending, find latest version, generate new version as latest + 1
    iii. If not closed or closed pending, previous checkin was incomplete, so keep same version
    iv. If previous checkin is closed or there is no previous checkin, create new db element, add to db with new key (see d)
    v. Otherwise, edit existing DbElement for open checkin (there can only be one)
d. Define unique key, i.e., namespace::filename.ext.ver
e. Create (new version) or update (open checkin) metadata
    i. Copy author, existing child keys, description from previous version if new version, or create from user supplied checkin information for new item, e.g., ver 1.
    ii. Update any changes in metadata specified in checkin information
f. Append version number to file
g. Submit file to StorageMgr for saving
h. Close checkin if requested
    i. May need to set as close pending
i. Notify client of successful checkin (open, closed pending, or closed)

Note: Do Not Close checkins for child keys !!!
Note: If you make a task list like this, implementing the process is very straight-forward.

16. 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.

C++ Syntax and Semantics

C++ is a large, powerful, and, in many ways, elegant language for building software components, applications, and systems.

```
While(true)
    Read a topic
    Write code relevant to that topic
    Choose another topic
End While
```

17. 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.

18. 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[6]. 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[7].

    Since **all other** methods used by the derived class are inherited[8], 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.

[6] 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.

[7] It 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.

[8] 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.

19. 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)[9]. 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.

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

20. 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.

Extendable and Reusuable Code Designs

C++, especially C++11 and later editions, afford you the opportunity to build flexible, robust, and elegant code. You do that using Templates, Inheritance, Interfaces, Object factories, and some inspiration.

21. Write all the code for a File Manager that searches a directory tree, rooted at a specified path. passing file names and directory names to a using application[10], without including any application details.

```
template<class App>
class FileMgr
{
public:
  using Patterns = std::vector<std::string>;
  using Files = std::vector<std::string>;
  using Dirs = std::vector<std::string>;

  bool search(const std::string& path);
  void addPattern(const std::string& pattern)  // not required for answer
  {
    patterns_.push_back(pattern);
  }
private:
  std::vector<std::string> patterns_;
};

template<typename App>
bool FileMgr<App>::search(const std::string& path)
{
  if (!FileSystem::Directory::exists(path))
  return false;
  App app;
  std::string fullPath = FileSystem::Path::getFullFileSpec(path);
  app.doDir(fullPath);

  for (auto patt : patterns_)
  {
    Files files = FileSystem::Directory::getFiles(fullPath, patt);
    for (auto file : files)
      app.doFile(file);
  }
  Dirs dirs = FileSystem::Directory::getDirectories(path);
  for (auto dir : dirs)
  {
    if (dir == "." || dir == "..")
      continue;
    std::string fullDir = fullPath + "\\" + dir;
    search(fullDir);
  }
  return true;
}
```

```
// Pseudo Code:
//   In class FileMgr<App>
//   Search(path)
//      App.doDir(path)
//      Find files
//      For each file
//          App.doFile(file)
//       Find Dirs
//       For each dir
//           Search(path + /dir)
//       return
```

[10] You may assume you have a FileSystem package with methods:
std::vector<std::string> FileSystem::getFiles(const std::string& path, const std::string& pattern)
std::vector<std::string> FileSystem::getDirectories(const std::string& path);

22. 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.

```
//----< 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;
  }
}
```

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

23. Write all the code for a class that wraps any C++ primitive[11] to enable instances of the class to behave like the primitive[12], 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();
  }
};
}
```

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

24. 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[13].

```
// 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.

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

Threads and Threading Models

Threads are very useful. We use them to:

a. Prevent a Graphical User Interface from freezing when the application needs to do some intense CPU activity. Allocating that work to a child thread allows the main GUI thread to continue handling user events.
b. Prevent a communcation channel from blocking an application's activities, waiting for transmission of messages. Use of sending and receiving threads and queues accomplishes this very effectively.
c. Fully use all of the CPU cores available for CPU intensive activities.
d. Avoid waiting for very slow human reactions, and I/O blocking, by doing background work.

25. Given a function with declaration:

    Result meaningOfLife(DateTime dt);[14]

    where result contains a size_t value and a std::string, write code to run the method asynchronously.

```
struct Result
{
  size_t mol;
  std::string blahBlah;
};

//////////////////////////////////////////////////////
// function definition is not required for answer
Result meaningOfLife(DateTime dt)
{
  std::this_thread::sleep_for(std::chrono::milliseconds(2000));
  Result result;
  result.mol = 42;
  result.blahBlah = "The meaning of life on " + std::string(dt) +
    " is ,as any fool can see, ";
  return result;
}
//////////////////////////////////////////////////////

int main()
{
  std::cout << "\n  MT1Q7 - Meaning of life";
  std::cout << "\n ========================\n";

  std::future<Result> fut = std::async(meaningOfLife, DateTime().now());
  Result result = fut.get();
  std::cout << "\n  " << result.blahBlah << result.mol << "!\n";
  std::cout << "\n  Thanks, Douglas Adams, for all the fish.\n\n";
  return 0;
}

//////////////////////////////////////////////////////////////////////////////////
 Definition of struct Result and the 3rd and 4th lines of main are a complete answer.
//////////////////////////////////////////////////////////////////////////////////
```

[14] You don't need to specify the body of the function, but will need to show the definition of Result, to fully answer this question.

26. Write all the code for a function that searches for a string of text in a text file, asynchronously[15]. 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.

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

27. 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.

```
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;
}
```

No code is required for this question.

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 .

28. 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.