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. Describe all the standard C++ relationships between classes. How did you use each of them in your design of Project #2?

   Answer:

   The C++ class relationships are:

   - **Inheritance** – publically derived classes inherit the public interface and implementation of the base class with the exception of constructors, assignment operator, and destructor. Those, however, are generated by the compiler if not declared in the derived class. Inheritance was used in Parser, e.g., classes derived from IRule, IAction, IBuilder, and ITokenCollection.

   - **Composition** – The composer holds an instance of the composed class as a data member. The C++ language guarantees that constructors will be called for the composed as the first act of the composer constructor. Thus, the composed is always an integral part of the composer, and, in fact, lies within the memory foot print of the composer. The Parser's Repository holds an instance of a ScopeStack as a data member.

   - **Aggregation** – The aggregating class may hold a pointer typed to hold an instance of the aggregated class and may eventually create a new instance bound to the pointer. Alternately the aggregator may create an instance of the aggregated class as local data in one of its member functions. The Parser's builder, ConfigureParser, holds references of many aggregated objects, e.g., every derived rule and action as well as the tokenizer and a semiexpression instance.

   - **Using** – a using class does not own its used instance. That must be provided by some other code and passed as an argument to a member function of the using class. The Parser uses all it's rules, owned by the ConfigureParser instance.

   - **Friend** – a class may grant another class or function access to its private data through a friend statement. We should avoid this unless there is no other way to meet the class's requirements. (No loose of points if you don't mention this.)

2. Write all the code for a Scope class that holds type, name, complexity, and an ordered collection of child scope types. Now write a matching function using all this information and uses a tolerance on complexity values. That would support matching when the complexities are close but not equal. The function should return true/false values.

   Answer:

```
class Scope
{
public:
  std::string& name() { return _name; }
  std::string& type() { return _type; }
  size_t& complexity() { return _complexity; }
  std::vector<std::string>& childType() { return _childType; }

  static bool match(Scope* pS1, Scope* pS2);

private:
  std::string _name = "unknown";
  std::string _type = "unknown";
  size_t _complexity = 0;
  std::vector<std::string> _childType;  // child scope types must be added in
};                                      // order encountered in parent scope

inline bool Scope::match(Scope* pS1, Scope* pS2)
{
  if (pS1->type() != pS2->type())
    return false;
  size_t complexityTolerance = 1;
  bool complexityMatch =
    (pS1->complexity() < pS2->complexity() + complexityTolerance) &&
    (pS1->complexity() > pS2->complexity() - complexityTolerance);

  bool childTypeMatch = true;
  if (pS1->childType().size() == pS2->childType().size())
  {
    for (size_t i = 0; i < pS1->childType().size(); ++i)
    {
      if (pS1->childType()[i] != pS2->childType()[i])
      {
        childTypeMatch = false;  // assumes each child type added in
        break;                   // order encountered in  parent scope
      }
    }
  }
  else
  {
    childTypeMatch = false;
  }
  return childTypeMatch && complexityMatch;
}
```

3. Which parts of a C++ class are not inherited? Why do you think the language was designed this way?

   Answer:

   Constructors, assignment operator, and destructor are not inherited, but will be generated if the class does not declare them and are needed by code that uses the class. Of course, generated methods are only correct if all the class's bases and member instances have correct construction, assignment, and destruction semantics.

   It would not make sense for these to be inherited since derived classes usually declare instances of data members and so inheriting these methods from the base class would not provide correct semantics as the derived data would not be initialized, assigned, and destroyed.

4. State the Dependency Inversion Principle and describe how you have used it in Project #2. If you have not used it, describe a useful way you could use it for that project.

   Answer:

   The Dependency Inversion Principle (DIP) states that software entities should depend on abstractions, not on implementation details. That means that we use interfaces and object factories to provide client access without binding to any implementation detail.

   The parser is a good example application of DIP. The ConfigureParser builder is an object factory for derived rules and actions. The parser accesses its rules with IRule pointers and the rules access their actions with IAction pointers.

5. Write all the code for a component that finds a common substring in two input strings if it exists. If so it returns the starting and ending positions for each input string. Given two strings it should return a Boolean value to note the presence or absence of a substring and support returning positions as a separate call. Note that this question focuses on the component structure and does not expect you to develop the fastest way to do this comparison.

   Answer:

```
struct ICompString  // component part
{
  using Interval = std::pair<size_t, size_t>;
  using Positions = std::pair<Interval, Interval>;

  virtual ~ICompString() {}
  static ICompString* Create();
  virtual bool compare(const std::string& s1, const std::string& s2, size_
t matchLength) = 0;
  virtual Positions locate() = 0;
};

class CompString : public ICompString  // comparison part
{
public:
  bool compare(const std::string& s1, const std::string& s2, size_t matchL
ength);
  Positions locate();
private:
  Interval Interv1{ 0, 0 };
  Interval Interv2{ 0, 0 };
};

ICompString* ICompString::Create() { return new CompString; }
```

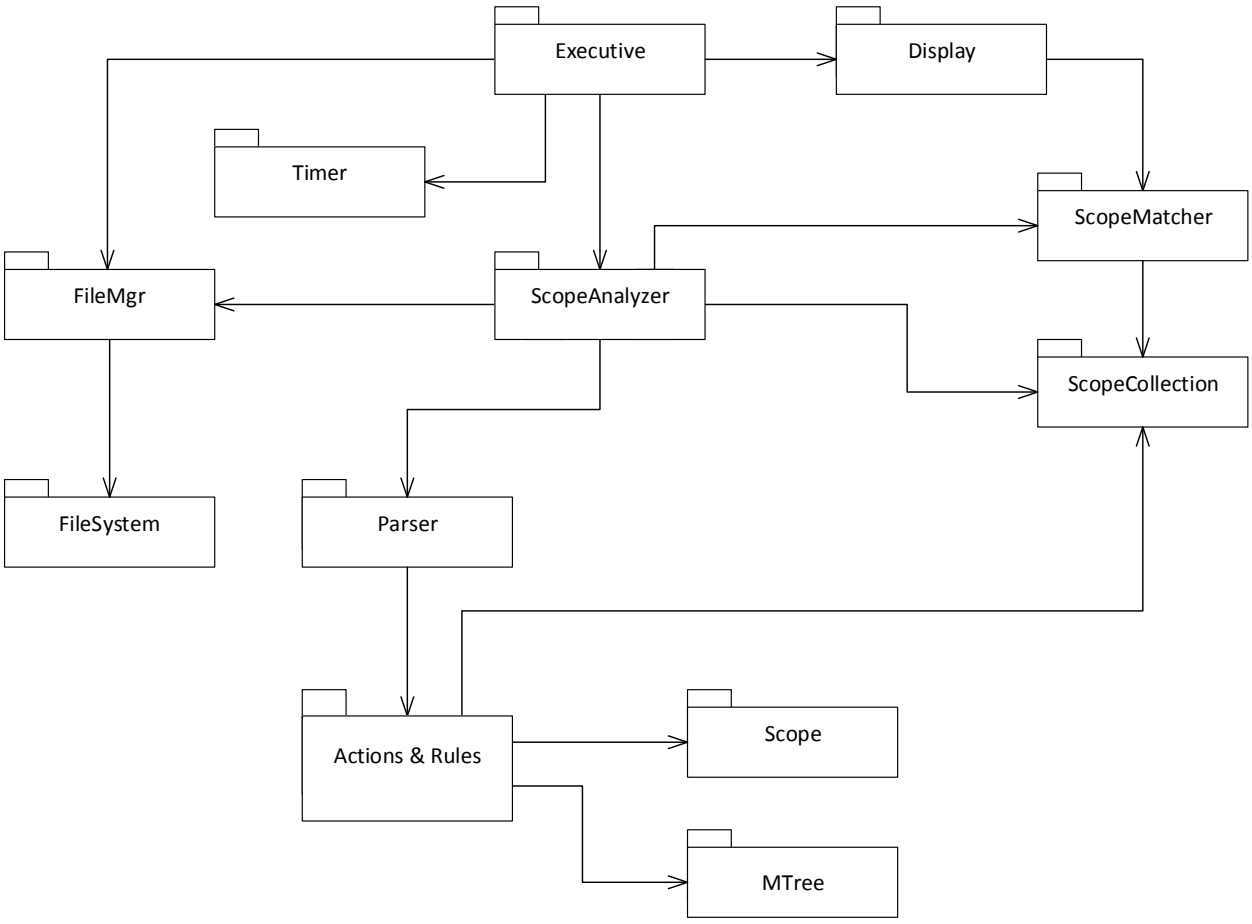
```
//----< find common substring >---------------------------------

bool CompString::compare(const std::string& s1, const std::string& s2, siz
e_t matchLength)
{
  bool match = false;
  for (size_t i = 0; i < s1.size() - matchLength + 1; ++i)
  {
    for (size_t j = 0; j < s2.size() - matchLength + 1; ++j)
    {
      if (s1[i] == s2[j])
      {
        for (size_t k = 0; k < std::min(s1.size(), s2.size()); ++k)
        {
          if (s1[i + k] != s2[j + k])
          {
            if (k >= matchLength)
            {
              Interv1.first = i;
              Interv1.second = i + k - 1;
              Interv2.first = j;
              Interv2.second = j + k - 1;
              return true;
            }
            else
              break;
          }
        }
      }
    }
  }
  return false;
}

CompString::Positions CompString::locate()
{
  Positions pos;
  pos.first = Interv1;
  pos.second = Interv2;
  return pos;
}
```

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

   Answer:



The Executive creates FileMgr, ScopeAnalyzer, and Display. It then starts its timer and requests FileMgr to build a file specification set for analysis, then starts ScopeAnalyzer. When ScopeAnalyzer returns it stops its timer.

ScopeAnalyzer retrieves a file specification from FileMgr and passes to Parser for analysis. Parser's actions build an MTree<MNode<Scope>> in the Repository and add references for each MNode<Scope> in the ScopeCollection. This process is repeated until there are no more files to analyze.

When file analysis is complete ScopeAnalyzer sorts the ScopeCollection and requests ScopeMatcher to find match candidates, which it may explore further using its access to the MTree, via the Repository. When completed it returns to the Executive which then requests Display to show the results.

7. Describe the C++ compilation model and how it affects your implementation of Project #2.

   Answer:

   The C++ compilation model consists of several phases:

   - **Preprocessing** replaces macros with their definitions, #include directives with the included header text, and includes or excludes source text based on #ifdef and #ifndef directives.
   - **Compilation** operates on one compilation unit, e.g., a preprocessed cpp implementation file, at a time, generating obj files for each.
   - **Linking** binds the obj files and any specified libraries into an executable exe file.
   - **Loading** binds executable with any specified dynamic link libraries to assemble the final image in memory.

   The C++ compilation model encourages us to define separate packages for each program activity which are then built into an executable whole. We manage the compilation sequence with preprocessor directives and can build the project in parts for incremental testing.

   In Project #2 we define several packages each of which focuses on a single activity like file management, parsing, scope analysis, and display. Each package that depends on the services of another package must include the header file of the used package, e.g., #include "service.h". For any standard libraries that are used by the package there must be a directive: #include <libfile>.

   If you have elected to build some of the program's services as Dynamic Link Libraries, your project must contain the Lib file that describes the DLL exports.