Swinburne University of Technology

Faculty of Science, Engineering and Technology

LABORATORY COVER SHEET

Subject Code:	COS30008
Subject Title:	Data Structures and Patterns
Lab number and title:	9, Abstract Data Types & Design Patterns
Lecturer:	Dr. Markus Lumpe

My life seemed to be a series of events and accidents.
Yet when I look back, I see a pattern.

Benoît B. Mandelbrot

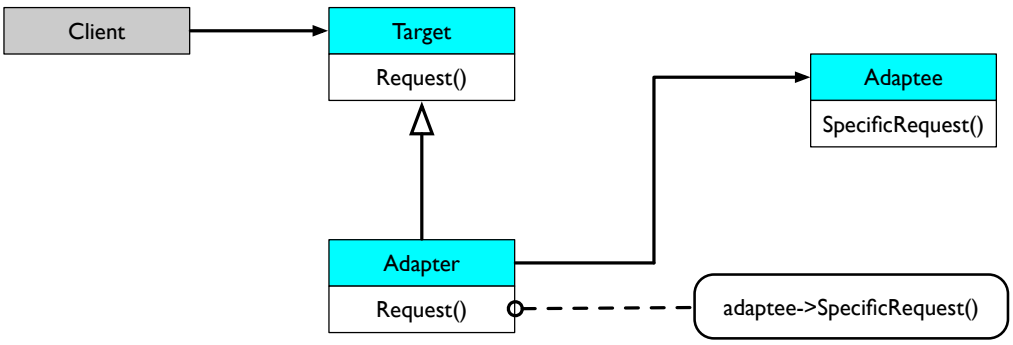

Figure 1: Object Adapter Design Pattern.

Problem 1

In this tutorial, we construct an object adapter for `std::ofstream` objects to write 12-bit values to files. The corresponding file output stream must be in binary format**,** and the data will consist of strings of `0`s and `1`s. The challenge is that we must use an 8-bit stream to write 12-bit values, which cannot be done directly. Instead, we need to employ a buffering mechanism to "collect" the bits, and once the buffer is full, we write it *en bloc* to the underlying 8-bit output stream.

12-bit values occupy 1½ bytes. Yet the smallest storage unit is a byte, or 8 bits. We need to devise an algorithm that seamlessly allows for both the output of 12-bit values to a buffer of bytes and the clearing (that is, writing data to file) of the buffer of bytes intermittently even when a write operation for a 12-bit value is still in progress, that is, the 12-bit value has only been partly written to the buffer of bytes. We must develop a plan and a viable strategy to cope with either scenario.

A suggested specification for a 12-bit output stream class is shown below:

```
#include <cstddef>
#include <fstream>


class ofstream12
{
private:
  std::ofstream fOStream;

  std::byte* fBuffer;            // output buffer
  size_t fBufferSize;            // output buffer size

  size_t fByteIndex;             // current byte index
  int fBitIndex;                 // current bit index (can be negative)

  void init();                   // initialize data members
  void completeWriteBit();       // complete write
  void writeBit0();              // write 0
  void writeBit1();              // write 1

public:
  // using C++11's nullptr
  ofstream12( const char *aFileName = nullptr, size_t aBufferSize = 128 );
  ~ofstream12();

  void open( const char *aFileName );
  void close();

  bool good() const;
  bool is_open() const;

  void flush();

  ofstream12& operator<<( size_t aValue );
};
```

Class `ofstream12` constitutes an object adapter. An object adapter maintains a *delegate instance* that performs the actual operations. The object adapter defines a *wrapper* that maps the required functionality (here, 12-bit I/O) to the provided functionality (here, 8-bit I/O). The object adapter should adhere to the service interface of the delegate instance, but may differ in some details. Here, we focus on the stream and file facets.

Class `ofstream12` requires a constructor and a destructor. The constructor must initialize the object, allocate the necessary buffer memory, and open the output file. The destructor must flush the buffer, close the underlying file, and release the buffer memory. Refer to

previous tutorials for details on allocating and freeing memory. Please note that we do not intend to create subclasses of `ofstream12`; therefore, there is no virtual destructor.

The `open`, `close`, `good`, and `is_open` methods correspond to their respective `std::ofstream` methods.

The method `flush` writes any pending output to the underlying output stream. We need to determine the actual number of bytes to be written. This can be subtle, as the true number of bytes that need to be written to the file can vary and may need to be adjusted by one at times.

Class `ofstream12` also defines an output operator as a member function. This is a valid approach, as the first argument (left-hand side of <<) is `this` object, which is an ofstream12 object. (Remember, `this` is a pointer to **`this`** object, whereas `*`**`this`** is *this object*.) The output operator implements the algorithm shown in this tutorial for writing 12-bit values. It uses the private member functions `writeBit0`, `writeBit1`, and `completeWriteBit`. These methods use the two indices `fByteIndex` and `fBitIndex` to perform the operations. The former refers to the current byte in the buffer a bit is written to, whereas the latter indicates the actual bit offset. Remember, we can only address bytes in C++. We must use bit operations to set or get bits of a byte. For example, to set bit 6 in a byte, we can use the expression `1 << 6`. Bit indices run from 0 to 7. When referring to bits from left to right, we start at 7 and continue down to 0. The method `completeWriteBit` performs the housekeeping. It triggers a flush when necessary and updates the byte indices.

You can use the following main function to test your code:

```
void write4096()
{
  cout << "Write 4096 codes" << endl;

  ofstream12 lWriter( "sample.lzw" );

  if ( !lWriter.good() )
  {
    cerr << "Error: Unable to open output file!" << endl;
    exit( 1 );
  }

  for ( size_t i = 4096; i > 0; )
  {
    lWriter << --i;
  }
}

int main()
{
  write4096();

  cout << "SUCCESS" << endl;

  return 0;
}
```

The output file `sample.lzw` contains binary data that cannot be viewed with a text editor. Use a hex editor or the hex view feature in Visual Studio or XCode. Google provides the right answers here. The size of the file `sample.lzw` is 6,144 bytes.

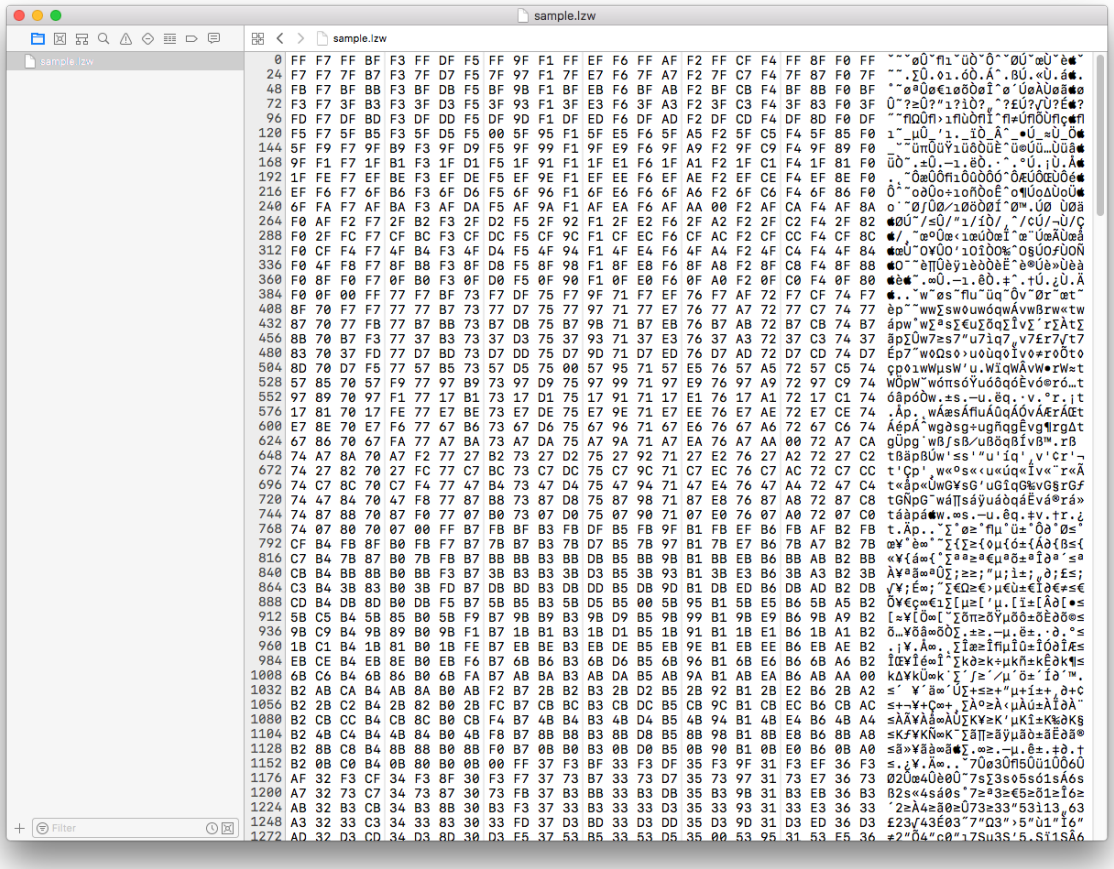

Figure 2: Hex view in XCode

Please check with the tutor. You should complete this task as it is a prerequisite for a later one.