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:	7, Design Pattern
Lecturer:	Ms. Siti Hawa

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

Benoît B. Mandelbrot

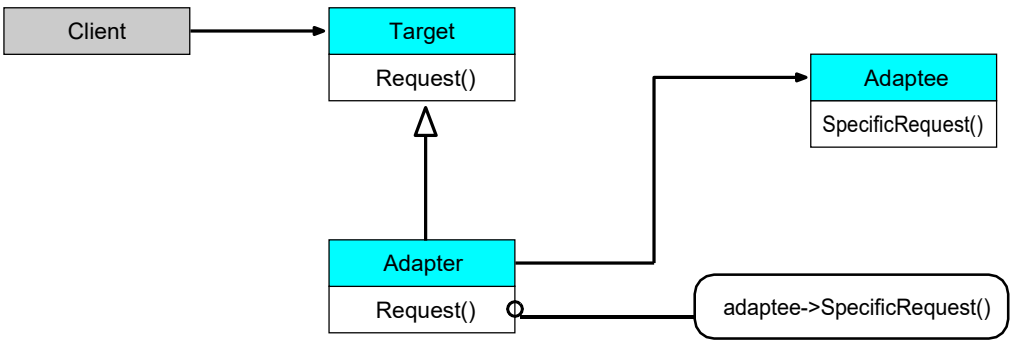

Figure 1: Object Adapter Design Pattern.

Problem 1

The goal of this tutorial session is to construct an object adapter for `ofstream` objects that allows us to write 12-bit values to files. Clearly, the corresponding file output stream has to be **binary** and the data would constitute strings of `0`s and `1`s. What makes this task somewhat difficult is the requirement that we need to use a physical 8-bit stream to write 12-bit values. We cannot do this directly. Instead, we have 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, that is, 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 a full buffer to output) 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 need to develop a plan and a viable strategy to cope with either scenario.

The object adapter, we wish to construct is given by the following class specification:

```
#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 isOpen() 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 provides 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 to 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 has to initialize the object, acquire the necessary buffer memory, and open the output file. The destructor has to flush the buffer, close the underlying file, and free the buffer memory. See tutorial 5

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

The methods `open`, `close`, `good`, and `isOpen` correspond to their respective `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 actual number of bytes that need to be written to file can vary, that is, 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 a ofstream12 object. (Remember, `this` is a pointer to `this object`, whereas `*this` is this object.) The output operator implements the algorithm shown 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++. To set or get bits of a byte we need to use bit operations. For example, the set bit 6 in a byte, we can use the expression `1 << 6`. Bit indices run from 0 to 7. Addressing bits in a byte requires us to start at 7 down to 0. The method `completeWriteBit` performs the house keeping. It triggers 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 simply viewed with a text editor. You either need to use a hex editor or use the hex view feature in Visual Studio or XCode. Google provides the right answers here. The size of file `sample.lzw` is 6,144 bytes.

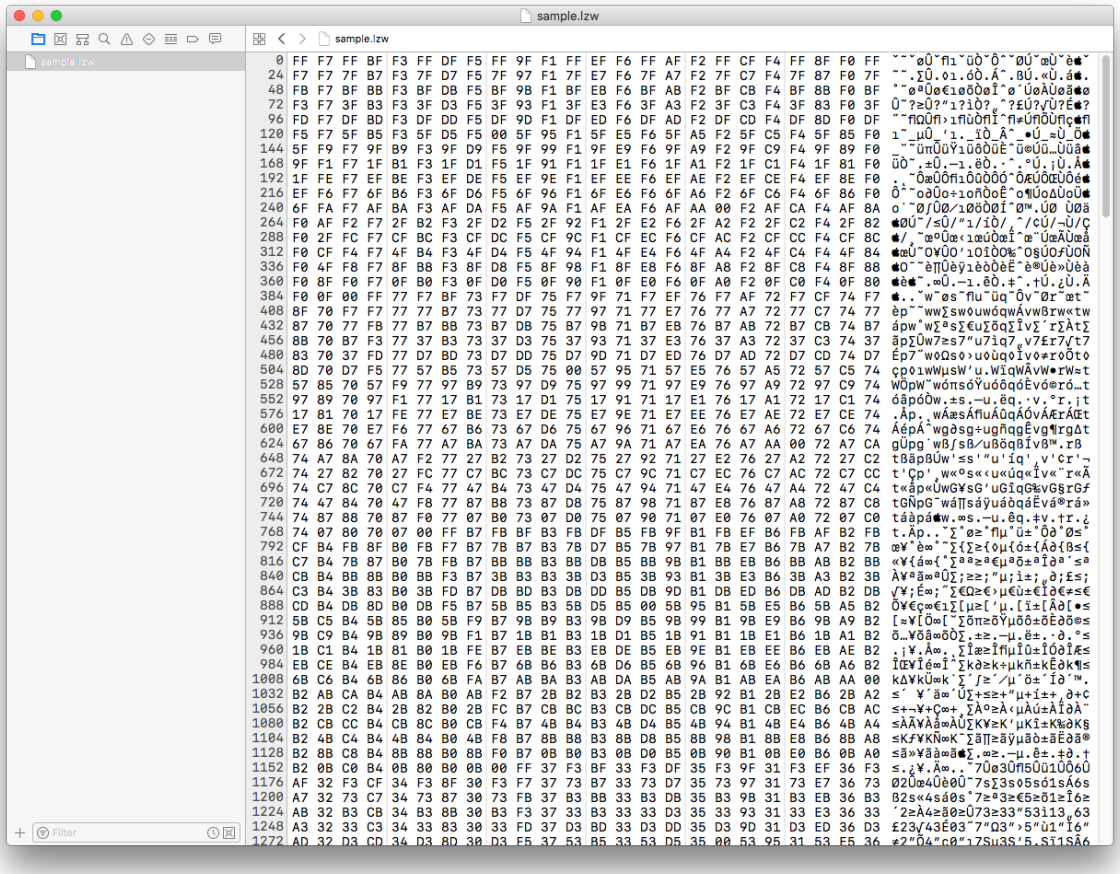

Figure 2: Hex view in XCode

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