CS 3450 – Design Patterns
Program 3

Date Due: See Syllabus
The usual submission requirements apply.

Write a number of classes that decorate a simple, generic stream output class. Here's the C++ interface that the stream output class and your decorators will implement:

```
#include <string>
template<typename T>
class Output {
public:
    virtual ~Output(){}
    virtual void write(const T&) = 0;
    virtual void writeString(const std::string&) = 0;

};
```

In Java it looks like this:

```
public interface Output {
    void write(Object o);

}
```

The string version of **write** is not necessary in Java since stream output automatically calls **toString** on objects.

The concrete class you will decorate is named **StreamOutput**. This class contains a reference to an output stream, and its **write** method expects a *streamable* argument (something that can be inserted into a C++ stream (via <<) or that has a **toString** method in Java – these are in the files that accompany this Zip file):

```
// StreamOutput.h
#include "Output.h"
#include <iostream>

template<typename T>
class StreamOutput : public Output<T> {
    std::ostream& sink;
public:
    explicit StreamOutput(std::ostream& stream) : sink(stream) {}
    void write(const T& t) {
        sink << t;

    }
    void writeString(const std::string& s) {
       sink << s;

    }
};

// StreamOutput.java
import java.io.*;
class StreamOutput implements Output {
    private Writer sink;
    public StreamOutput(Writer stream) {

        sink = stream;
    }

    public void write(Object o) {
        try {
            sink.write(o.toString());

        }
        catch (IOException ex) {
            throw new RuntimeException(ex);

        }
    }
}
```

Write the following decorators for these classes:

1. **LineOutput**: adds a newline with every write
2. **NumberedOutput**: this adds newlines, and precedes each write with the current line number (1-based) right-justified in a field of width 5, followed by a colon and a space.
3. **TeeOutput**: writes to two streams at a time; the one it wraps, plus one it receives as a constructor argument
4. **FilterOutput**: writes only those objects that satisfy a certain condition (unary predicate), received as a constructor parameter.

To do **FilterOutput**, you will probably want to use a template argument in C++, but you're welcome to follow the Java approach, which is to use interface like:

```
public interface Predicate {
    public boolean execute(Object o); // used by FilterOutput

}
```

You can use the file *decorator.dat* in the Zip file for testing. Here is sample output from a test that reads a line at a time from this file, and runs each line through **LineOutput** and a **FilterOutput** I created named **ContainsDigit**:

```
        _count = 0    # A shared value for Shape classes with no current
        objects cls._count += 1         # Create/update class attribute
```

Test a reasonably-sized chain of decorators. Note that decorators such as **LineOutput** and **NumberedOutput** are normally applied *after* **FilterOutput** decorators execute, because they change the output, which may affect the result of the filter's predicate. So filters are often applied upstream, meaning they should be the outermost wrappers. You might want to try it both ways to see what happens.

On the other hand, **TeeOutput** decorators should be applied downstream if you want the full result of all your decorators to go the tee destination (simply because nested decorators constitute a sequence of operations). Bottom line: order of composition of decorators matters!

Note that decorators are applied in the reverse order in which they are created (think Last in, First out), so the last decorator you create is the first one to decorate the line from the file.

Program Specification

The program should decorate the object dynamically:

1. When the program runs, it should prompt for a file to read. (It’s up to you whether to read the contents of the file at this time, or wait until the decorator chain is created. I think it’s
a lot easier to wait, then process the file a line at a time.)
2. It then presents a menu of the desired decorations to apply.
    a. The user can select multiple decorations, including multiple instances of the same type of decorator; they will be applied in order of selection.
    b. If the user selects TeeOutput, prompt for a file name to direct the Tee output to.
    c. If the user selects FilterOutput, give the user a choice of at least two predicates to choose from.
3. Then produce the decorated output.

Programming note: you are creating the Decorator objects right in your menu code – doing it any other way is both a lot harder and usually wrong!

Note to C++ programmers: the **ostringstream** class in <sstream> is handy for this assignment.

Note to everyone: you should derive the decorators from the base class, not the OutputStream class (think about why.)