

## Introduction to Design Patterns

---

See also: Chapter 4 (102-103), Chapter 26 (753-760)

### **Design Patterns? That's what we did in my quilting class last quarter**

From the textbook:

A design pattern is a standard approach to program organization that solves a general problem. C++ is an object-oriented language, so the design patterns of interest to C++ programmers are generally object-oriented patterns, which describe strategies for organizing objects and object relationships in your programs. These patterns are usually applicable to any object-oriented language, such as C++, Java, or Smalltalk. In fact, if you are familiar with Java programming, you will recognize many of these patterns.

Design patterns are less language specific than are techniques. The difference between a pattern and a technique is admittedly fuzzy, and different books employ different definitions. This book defines a technique as a strategy particular to the C++ language that overcomes a deficiency in the language itself, while a pattern is a more general strategy for object-oriented design applicable to any object-oriented language.

Note that many patterns have several different names. The distinctions between the patterns themselves can be somewhat vague, with different sources describing and categorizing them slightly differently. In fact, depending on the books or other sources you use, you may find the same name applied to different patterns. There is even disagreement as to which design approaches qualify as patterns. With a few exceptions, this book follows the terminology used in the seminal book *Design Patterns: Elements of Reusable Object-Oriented Software*, by Erich Gamma et al. However, other pattern names and variations are noted when appropriate.

### **The Singleton Pattern**

The *singleton* is one of the simplest design patterns. In English the word "singleton" means "one of a kind" or "individual." It has a similar meaning in programming. The singleton pattern is a strategy for enforcing the existence of exactly one instance of a class in a program. Applying the singleton pattern to a class guarantees that only one object of that class will ever be created. The singleton pattern also specifies that the one object is globally accessible from anywhere in the program. Programmers usually refer to a class following the singleton pattern as a *singleton class*.

## The Singleton Logger – using static

```
/**
 * Logger.h
 *
 * Definition of a singleton logger class, implemented with static methods.
 */

#include <iostream>
#include <fstream>
#include <vector>

class Logger
{
public:
    static const std::string kLogLevelDebug;
    static const std::string kLogLevelInfo;
    static const std::string kLogLevelError;

    // Logs a single message at the given log level
    static void log(const std::string& inMessage,
                   const std::string& inLogLevel);

    // Logs a vector of messages at the given log level
    static void log(const std::vector<std::string>& inMessages,
                   const std::string& inLogLevel);

    // Closes the log file
    static void teardown();

protected:
    static void init();

    static const char* const kLogFileName;

    static bool sInitialized;
    static std::ofstream sOutputStream;

private:
    Logger() {}
};
```

```
/**
 * Logger.cpp
 *
 * Implementation of a singleton logger class.
 */

#include "Logger.h"

using namespace std;

const string Logger::kLogLevelDebug = "DEBUG";
const string Logger::kLogLevelInfo = "INFO";
const string Logger::kLogLevelError = "ERROR";

const char* const Logger::kLogFileName = "log.out";
```

```

bool Logger::sInitialized = false;
ofstream Logger::sOutputStream;

void Logger::log(const string& inMessage, const string& inLogLevel)
{
    if (!sInitialized) {
        init();
    }
    // print the message and flush the stream with endl
    sOutputStream << inLogLevel << ": " << inMessage << endl;
}

void Logger::log(const vector<string>& inMessages, const string& inLogLevel)
{
    for (size_t i = 0; i < inMessages.size(); i++) {
        log(inMessages[i], inLogLevel);
    }
}

void Logger::teardown()
{
    if (sInitialized) {
        sOutputStream.close();
        sInitialized = false;
    }
}

void Logger::init()
{
    if (!sInitialized) {
        sOutputStream.open(kLogFileName, ios_base::app);
        if (!sOutputStream.good()) {
            cerr << "Unable to initialize the Logger!" << endl;
            return;
        }
        sInitialized = true;
    }
}

```

## The Singleton Logger – using access control

```
/**
 * Logger.h
 *
 * Definition of a true singleton logger class.
 */
#include <iostream>
#include <fstream>
#include <vector>

class Logger
{
public:
    static const std::string kLogLevelDebug;
    static const std::string kLogLevelInfo;
    static const std::string kLogLevelError;

    // Returns a reference to the singleton Logger object
    static Logger& instance();

    // Logs a single message at the given log level
    void log(const std::string& inMessage,
             const std::string& inLogLevel);

    // Logs a vector of messages at the given log level
    void log(const std::vector<std::string>& inMessages,
             const std::string& inLogLevel);

protected:
    // Static variable for the one-and-only instance
    static Logger sInstance;

    // Constant for the file name
    static const char* const kLogFileName;

    // Data member for the output stream
    std::ofstream mOutputStream;

private:
    Logger();
    ~Logger();
};

/**
 * Logger.cpp
 *
 * Implementation of a singleton logger class.
 */
#include "Logger.h"

using namespace std;

const string Logger::kLogLevelDebug = "DEBUG";
const string Logger::kLogLevelInfo = "INFO";
const string Logger::kLogLevelError = "ERROR";
```

```

const char* const Logger::kLogFileName = "log.out";

// The static instance will be constructed when the program starts and
// destructed when it ends.
Logger Logger::sInstance;

Logger& Logger::instance()
{
    return sInstance;
}

Logger::~Logger()
{
    mOutputStream.close();
}

Logger::Logger()
{
    mOutputStream.open(kLogFileName, ios_base::app);
    if (!mOutputStream.good()) {
        cerr << "Unable to initialize the Logger!" << endl;
    }
}

void Logger::log(const string& inMessage, const string& inLogLevel)
{
    mOutputStream << inLogLevel << ": " << inMessage << endl;
}

void Logger::log(const vector<string>& inMessages, const string& inLogLevel)
{
    for (size_t i = 0; i < inMessages.size(); i++) {
        log(inMessages[i], inLogLevel);
    }
}

```