Patterns for Logging Diagnostic Messages

Neil B. Harrison
Bell Laboratories
11900 N. Pecos St.
Denver, CO 80234
(303) 538-1541
nbh@dr.att.com

Abstract

Every software system must deal with errors. Most systems report errors to the user in some manner, and many provide additional diagnostic information to assist the user in tracking down the problem. Transaction-oriented systems lend themselves to common approaches to logging diagnostic messages. These approaches are embodied in three general software patterns. The first pattern, Diagnostic Logger, separates logging from the rest of the software, and lays the groundwork for the other patterns. The second pattern, Transactional Buckets, provides association of diagnostics with the correct transactions. The third pattern, Typed Diagnostics, helps ensure uniformity of presentation for all diagnostics. It also allows the software to handle various diagnostics differently, depending on characteristics such as severity.

Introduction

Many software systems are transaction-oriented, that is, they take discrete, sometimes independent inputs, perform some operations, and generate some output. In business applications, programs deal with payroll or other personnel functions. Most database applications are query-response based. And in software development, compilers and other translators take input such as programs, and produce output such as object code.

All these systems must be able to handle error conditions. These may come about from bugs in the software, but often simply arise from erroneous input. In interactive systems, the system can give immediate feedback to the user that an input error has occurred. Ward Cunningham's Checks [Cunn95] pattern language is useful in this situation. However, if the input is batched, (for example, source code for a program is a batch of lines of code), the problem becomes one of producing diagnostic messages with sufficient context. Imagine trying to compile a program if the only message the compiler gives is "syntax error", with no line number!

A related issue is that although transactions themselves may be somewhat independent, processing a transaction may impact the processing of later transactions. At the very least, many programs keep track of the number of errors encountered, and give up when a certain threshold is reached. In some systems, certain errors may happen repeatedly, such as when a system keeps trying to re-establish a connection to another machine. So the message is given once, with periodic status messages (e.g. "The above message repeated 10 times"). So it is necessary for many systems to remember information about errors after they happen.

These issues are addressed in three following patterns. They are Diagnostic Logger, Transactional Buckets, and Typed Diagnostics. Each builds on the previous pattern.

Pattern: Diagnostic Logger

Problem:

How do you report diagnostic information in a consistent manner?

Context:

Software systems all have the need to provide feedback information to the user in case of errors or other difficulty. This pattern applies specifically to systems where users do not actively interact on a per-transaction basis, e.g., batch mode processing, compiling, and other types of translation systems.

Forces:

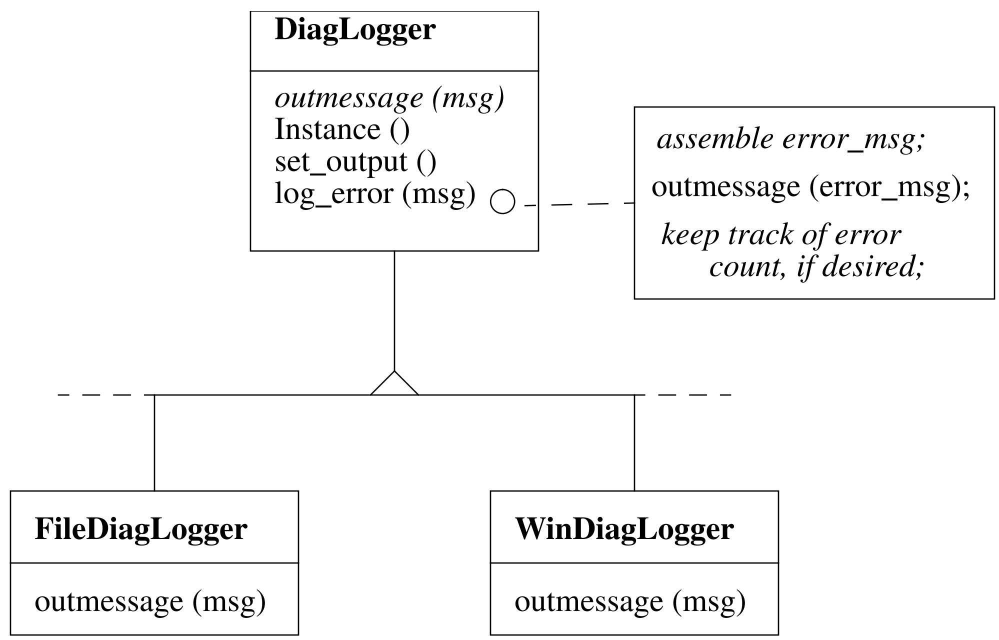
1. Any part of a system might have need to create a diagnostic message; therefore, any approach must be available to all parts of the system.
2. Diagnostic messages are very important to the users; they should have a consistent look and feel.
3. In most projects, error handling is largely ignored until coding is underway, and programmers realize that errors will happen. At this point, the programmers are most interested in simple, easy to use error reporting strategies. There is a strong temptation to throw something together with no regard to other programmers.
4. The user may wish to specify the destination of diagnostic messages.
5. The order of messages may be significant.
6. Diagnostic messages almost always have specific information; for example, a compiler may report the line number and suspected offending token where a syntax error occurs.
7. It may be important to retain information from one message to the next. For example, it is common to keep track of total errors, and exit when a threshold is crossed.

Solution:

Create a Diagnostic Logger object to handle all the details of diagnostics. Create it as a Singleton [Gamm94], so that there is a single point for all messages to flow through.

The Diagnostic Logger has two types of functions. The first type is for control of logging in general, such as specifying the output destination, error thresholds, or debugging levels. These are used chiefly by some main controlling objects. The other type of functions provides the rest of the software the ability to output the diagnostic messages. These message types may be as general as passing a message, but are usually broken down by type of message or event. In most cases, the Diagnostic Logger will provide an individual function for each type of messages, such as errors and warnings.

In order to handle different methods of output, it may be desirable to derive different types of Diagnostic Loggers. In such a case, the program specifies the destination of the output. The correct type of Diagnostic Logger is created either at that time or at the first time the instance() function is called. This is somewhat in the flavor of the Abstract Factory [Gamm94] pattern. In this case, the message functions may be implemented using the Template Method [Gamm94] pattern.



Resulting Context:

The Diagnostic Logger pattern provides consistency in diagnostic messages, with the details of outputting the messages encapsulated and hidden from the rest of the program. Because it is a Singleton, it easily preserves the order of the messages. The messaging functions are readily accessible to all parts of the system, and is easy to use. In fact, the interface can be decided on early in design, without worrying about the details of format or destination of messages. Developers may start out with rudimentary functionality in the Diagnostic Logger, such as dumping all messages to `stderr`.

Rationale:

This pattern began in a program which translated test cases to automatically executable test scripts. The program was later easily modified to generate a different test language, with the changes made inside the Diagnostic Logger. The Diagnostic Logger also sent diagnostic messages to a designated destination, while creating a different file for every test script it generated. This brings us to the next pattern, Transactional Buckets.

Pattern: Diagnostic Transactions

Problem:

If an error occurs in processing a set of discrete inputs, how do you associate error messages with the correct input?

Context:

The system is transaction-oriented; that is, it processes a set of discrete inputs. The inputs are somewhat autonomous, although they need not be entirely independent of each other. However, it is important that results (i.e. outputs and errors) be identified with their inputs. Inputs are often processed in batches, so associating output with input is not simply responding to a single query. Examples are compilers, interpreters, or database query systems.

The results of the inputs are not necessarily independent, such as total error count. In some cases, such as compiling a program, the inputs are highly dependent on each other, but the diagnostic messages are linked to specific inputs.

You are using Diagnostic Logger to handle diagnostic messages.

Forces:

1. The user wants to know which output was produced by which input. This is particularly important with error or other diagnostic messages.
2. A single input may produce multiple output messages.
3. Internally, the place where an error occurs may be many function calls away from the information which identifies the input. For example, a syntax error of an input statement may be caught deep in the parser, far from the input line number.

4. The information which identifies the input may be complex; for example, an input may be identified by the line number, file name, and directory name. It would be undesirable to pass such information through multiple functions in order to enable error reporting. Think what would happen if it becomes necessary to report machine name as well as the above information; many functions would have to change.

Solution:

Use the Diagnostic Logger pattern, and augment it with Diagnostic Transaction objects. A Diagnostic Transaction is an object which exists for the life of processing a particular transaction, and provides unique identification of that transaction. Its birth marks the beginning of processing a transaction, and its death marks the end.

Obviously, the easiest approach to this is to take advantage of scoping to create and destroy Diagnostic Transactions. For example, when processing a set of lines:

```
for (int i = 1; i < num_lines; i++)
{
      DiagTrans     dt (i);
      // Process the input line, generating diagnostic messages as appropriate

}  // The Diagnostic Transaction automatically gets destroyed.
```

The constructor for the Diagnostic Transaction registers with the Diagnostic Logger, and its destructor unregisters.

```
DiagTrans::DiagTrans (int line_no) :
                    _curr_line_no (line_no)
{
       DiagLogger::instance()->sign_on (this);
}

DiagTrans::~DiagTrans ()
{
       DiagLogger::instance ()->sign_off ();
}
```

Likewise, the DiagLogger ensures that there is only one transaction processed at a time:

```
void
DiagLogger::sign_on (DiagTrans* dt)
{
       if (_curr_diag_trans != NULL)
       {
          // Bad; take appropriate action
       }
```

```
        _curr_diag_trans = dt;

        //  Write a message indicating the start of a transaction
}

void
DiagLogger::sign_off ()
{
        _curr_diag_trans = NULL;
}
```

With Diagnostic Transactions, the software still continues to log messages directly to the Logger. However, the Logger will always check whether a Diagnostic Transaction has registered with it, forcing the creation of Diagnostic Transactions. Note that this presents a slight problem at startup; there may be errors to be logged before the first transaction is processed. One approach to this is until the first time a Diagnostic Transaction registers with the Logger, the Logger simply outputs the messages, with no surrounding context.

Resulting Context:

The Logger can mark the start and end of transactions.

Diagnostic Transactions provide an easy way to implement leaky bucket counters. Because the Logger hears about every transaction, it can count successful or unsuccessful processing of transactions. On the other hand, Diagnostic Transactions can handle errors on a per transaction basis. For example, a Diagnostic Transaction may limit the number of errors from a single transaction, without aborting the entire session.

Variations:

It may be convenient to use a single DiagLogger and Diagnostic Transactions to implement diagnostic message handling in a multithreaded environment. In such a case, the DiagLogger would keep track of a Diagnostic Transaction for each thread. Each Diagnostic Transaction would be created with some information which would uniquely identify its thread. This is similar to the registry of singletons discussed in the Singleton pattern description.

In other cases, it may be desirable to allow nested Diagnostic Transactions to provide layers of contexual information. The DiagLogger would manage the stack of Diagnostic Transactions. This makes adding messages and new context easy; simply create a new Diagnoatic Transaction to capture the desired context.

Rationale:

The Diagnostic Transaction came about as a result of trying other, inferior approaches to associating messages with the transactions from which they came. Originally, it was necessary to register explicitly at the beginning and end of processing each transaction. The Diagnostic Transaction is a more automatic and safer approach to this problem.

Pattern: Typed Diagnostics

Problem:

If you are logging many different messages (such as errors, warnings, debugging messages, etc.), how do you ensure that these messages are handled consistently?

Context:

The system uses the Diagnostic Logger to handle diagnostic messages. There are several different types of events which must be handled consistently, but with enough variation to allow the essential information to be trapped and passed to the user. The system is a transaction-oriented system, and you are using the Diagnostic Transaction to associate messages with each other on a per-transaction basis.

Forces:

1. Once again, it is necessary to keep the details of handling messages out of the main program.
2. For each event you want to log, you want to capture the important information of the moment. Depending on the nature of the event, what is important may vary.
3. Handling diagnostic or debug messages can be troublesome. It is desirable to be able to simply call a diagnostic message handler member function of the Diagnostic Logger. However, diagnostics are usually turned off, but the user would still incur the cost of calling the message handler function. If character strings are created and passed as arguments to the function (which is likely), repeated calls to the function could create a noticeable performance impact.
4. In some cases, it may be desirable to handle messages differently depending on a later event. For example, an error report may include a record of previous events, but they would not normally be reported.

Solution:

Create an inheritance hierarchy of Diagnostic Messages. Each type encapsulates the characteristics of that category of diagnostics, and parameterizes the specific variations. In particular, details of diagnostic messages are embedded in the classes, so as not to incur overhead when creating the Diagnostic Message objects. In fact, the data in these classes should be pretty sparse, so as to minimize the overhead of creation and destruction.

It is necessary to send Diagnostic Messages to the Logger; this is usually accomplished at creation. The standard interface to the Logger for messages becomes a simple function for all types of diagnostics:

```
// For an error
DiagLogger::instance ()->message (new DiagMsg_Error(err_type, etc));

// For a debug message at debug level 2
DiagLogger::instance()->message (new DiagMsg_Debug (2, etc));
```

The Logger hands the Diagnostic Message to the Diagnostic Transaction, which then assumes the responsibility for managing them, including destruction of the Diagnostic Messages at the proper time.

Typed Diagnostic Messages combine with Diagnostic Transactions to handle conditional output of messages. Message handling can be quite flexible. For example, debugging messages would be conditionally output, depending on the current debugging level. On the other hand, context messages would be stored, and output in the event that an error message arrives.

Note that Typed Diagnostics are different than the previously discussed subclasses of Diagnostic Logger. Diagnostic Logger subclasses designate approaches to handling of all messages, such as output style or destination. Typed Diagnostics differentiate handling of different types of messages, such as error messages versus levels of debugging messages.

Resulting Context

With the application of Typed Diagnostic Messages, a program can output several different types of meaningful diagnostic messages, while still maintaining a consistent output style. It also retains the characteristics of Diagnostic Loggers, such as ease in changing output destination, and adaptation to different output types.

Note that Typed Diagnostic Messages do incur a performance cost. Objects may be created and later destroyed, even though they do not create output. For example, debugging Typed Diagnostic Messages are created and destroyed, but only used when debugging is turned on. Therefore, the classes should be designed so that creation and destruction are inexpensive. In practice, this has not been a problem. For message text, one might consider using a standard set of messages and message numbers.

Rationale

This pattern grew out of various attempts to create a way to handle different types of messages. In some cases, the output was automatically processed, so the messages had to be consistent. This was coupled with the desire to produce different levels of debug messages, but to avoid excessive overhead if debugging was not enabled.

Acknowledgments

The author wishes to thank Stephen Berczuk for his helpful comments, particularly for pointing out the benefits of nested Diagnostic Transactions.

References

[Cunn95] Cunningham, Ward. “The CHECKS Pattern Language of Information Integrity”, in *Pattern Languages of Program Design,* Addison-Wesley, Reading, Massachusetts, 1995, pp 145-155.

[Gamm94] Gamma, Erich et al. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, Massachusetts, 1995.