Chapter 1

An Extensible Object Logger

There are several logging frameworks available for Pharo: SimplerLogger, PaulThePoulpe and ToothPick. Toothpick is developed by Joseph Pelrine from MetaProg http://www.metaprog.com/Toothpick/. It is the most advanced and has a nice documentation. SimpleLogger developed by German Arduino is available at http://ss3.gemstone.com/ss/SimpleLogger.html. SimpleLogger is a really simple text only logger and Paul le Poulp is another interesting framework http://concretetypeinference.blogspot.fr/2012/06/pharo-logger-aka-paul-octopus-or-le.html.

So you can wonder why there is a need for yet another one. Well let us think a bit about the domain. Often logging is associated with outputting strings into a file and using grep to make sense out of them. Why only focusing on strings. We have objects and they can be converted into strings so we have only to win. SystemLogger is an easy to use, very lightweight, and highly configurable object logging framework. SystemLogger manipulates objects and this opens a complete new set of scenario and in addition it is compatible with the idea of a string logger.

1.1 Starting with an example

With SystemLogger logging a message is as simple as sending the message message: to the class Log.

```
Log message: 'Hello it looks we got a trouble'
```

Let us step back one minute. message: is equivalent to the old Transcript show:. Note however that what is cool with SystemLogger and dealing with objects is that you should not be concerned about the handling of carriage return. Why? Simply because log objects do not have this notion and when

we want to transform them into strings, then the formatter will do it for you. We are sure that you start understanding that dealing with objects is the way to go and will make our live easier.

When you are debugging and want to identify more specific paths in your code, issuing a message is a bit broad and it will be mixed with all the other messages. To help you finding your ways among the collected logs, you can also use the message trace:.

It looks simple and so far it is. What you see is that as a normal user of the log objects you just have to express what you want.Then the system takes care about collecting the logs into a logger as we will see later.

Now you may want to raise log of different importance. Log provides some predefined ways to express the severity of your log: error:, critical:, message:, trace: and warm:.

```
Log error: 'Damn this is broken'.
Log critical: 'Damn this is really broken'.
Log message: 'Hello it looks we got a trouble'.
Log trace: 'We passed here '.
Log warm: 'the system is still running but you should have look at it'.
```

Log creation messages

In addition to simple creation messages, Log proposes other creation messages to improve the filtering of logs.

The default creation messages are in addition to the simple message:.

- message: tag: where tag: is a domain to sort logs. It will help you
- message: tag: level: where level indicates an importance level. So far we have the following levels: critical error warning information trace that you can obtain doing LogLevel all.
- message: aMessage tag: aTag level: aLevel timeStamp: aTimeStamp. Note that normally you should not have to specify the time stamps of the log since this is done automatically.

Here is an example

```
(Log
      message: 'From My Framework'
      tag: 'Compiler'
      level: LogLevel error)
```

Extra information

A log can also accept any optional information using key/value

```
Log error: 'Damn this is broken' withExtra: [ :log | log extensionAt: #thisContext
    put:   thisContext]
Log message: 'Not totally broken' withExtra: [ :log | log addExtensions: {
    #thisContext -> thisContext} ]
```

Since we do not know the exact scenario the Log class proposes the following messages to handle extra information.

- addExtensions: aPair adds an extension and its value specified as a pair
- extensionAt: aSymbol and extensionAt: aSymbol ifAbsent: aBlock queries the value of an extension and execute the absent block if necessary.
- extensionAt: aSymbol put: aValue set the value of a key.
- extensionKeys returns the list of extensions.

1.2 Three core classes

SystemLogger core is composed about 3 classes: Log, SystemLogger and LogDispatcher . The two first ones are public while the last one is a private class that the end user does not have toas shown in Figure 1.1 know. It raises also announcements: LogAdded and LogRemoved.

In addition, LogLevel encapsulates the logic of defining a level. A logLevel represents a level of severity in log messages. There are default log levels which can be used via class methods such as critical, error, warning, information, debug and trace.

Core element one: Log objects

Log objects are represented by the classes Log and its superclass BasicLog. Log is named like that because we wanted to have it as short as possible to write concise code. Log represents a simple log entity: be it a compilation overrides, an image snapshot or a simple string output. It collaborates with the current log dispatcher to emit log objects to loggers.

Note that Log is named Log and not SLLogObject or whatever because it is the main entry point for a client point of view.

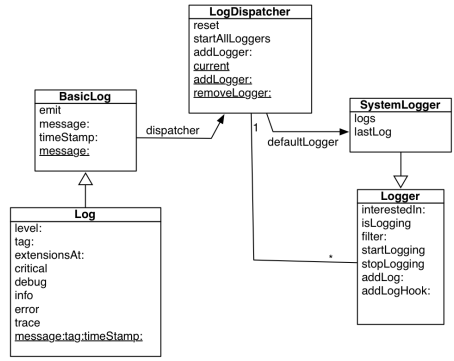

Figure 1.1: Core classes.

Important point.

One of the key responsibility of a log object is to emit the logger to the log dispatcher. However, in general you do it by yourself. The emit message is invoked for you by the creation messages. All the class creation methods use emit. The subclasses may introduce extra information such as policy to display, filtering, extra logging information. They should offer a nice API and invoke the emit message.

BasicLog, the superclass of Log represents a simpler log object consisting only of a timeStamp and a message. More specialized log classes may be derived from this class like the default Log class does. Usually as a normal user you will not use directly BasicLog but Log because it proposes a better API.

```
(BasicLog message: 'Test message string') emit
```

Core element two: LogDispatcher

When a log object is created, it is passed to the system log dispatcher. There is only one logDispatcher in the system, it is an instance of LogDispatcher and can be accessed LogDispatcher current. His role is to dispatch the emitted log objects to the system loggers that registered to the dispatcher.

There is one favorite system logger associated to the current dispatcher. It is the main default logger. It can be accessed by the following expression: self current defaultLogger.

In addition, we can add loggers to the current dispatcher using addLogger:. A dispatcher will dispatch log to any available loggers.

```
LogDispatcher current addLogger: (SystemLogger name: 'test1').
```

or

```
LogDispatcher addLogger: (SystemLogger name: 'test1').
```

Similarly you can remove a system logger using the removeLogger: or removeLoggerNamed: messages. Finally all loggers can be stopped to receive log objects from the log dispatcher and started using the messages startAllLoggers and stopAllLoggers.

Core element three: SystemLogger

The third player in the framework is a logger. A logger is responsible to decide if it is interested to handle log objects and if this is case to handle them according to its function.

Logger

Logger is the abstract superclass of system loggers. It has a name.

It defined the following messages:

- startLogging and stopLogging enable and respectively disable the handling of log objects.
- filter: aBlock to specify the condition that the log objects should satisfy to be handled by the system logger.
- addLog: aLog sent by the log dispatcher.
- addLogHook: aLog is a hook message that specifies the effective action performed when handling a log object.
- reset flushes the log objects if any that could have been hold by the receiver.

SystemLogger

SystemLogger is a subclass of Logger. It keeps the log objects in memory. One of its instance is the default system logger associated to the LogDispatcher singleton. We can imagine in the future to change it to support a cyclic structure to hold a maximum amount of log objects.

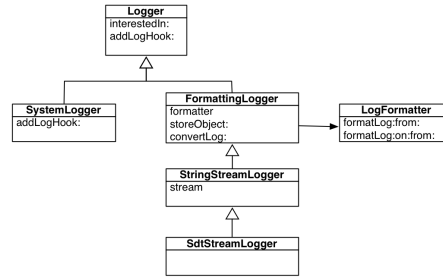

Figure 1.2: A first extension.

1.3 About LogLevels

As we saw you can raise logs of different severities: error:, critical:, message:, trace: and warm:. LogLevel represents such a severity. A log level that has a larger number for severity is more severe.

A convenience method LogLevel>>orMoreSevere allows one to express that we are interestedin a given level or more severe ones.

```
aLogger filter: LogLevel warning orMoreSevere
```

1.4 First useful extensions

The first extension is to provide a system logger that can emit strings to our lovely Transcript. The idea there is that in addition to log objects to the SystemLogger we would like that the systemcreate a textual trace in the Transcript as in the old days.

The package already defines it. Now we will simply explain some key aspects of the solution. StringStreamLogger is a special logger. It is a logger that emits formatted strings into a stream. It subclasses the class FormattingFormatter and add the notion of stream as show in Figure 1.2.

```
FormattingLogger subclass: #StringStreamLogger
   instanceVariableNames: 'stream'
   classVariableNames: ''
   category: 'SystemLogger−Extensions−StringOutputter'
```

The class FormattingLogger is a subclass of Logger (and not SystemLogger) that simply pass the logs object to a formatter before storing them.

```
FormattingLogger>>addLogHook: aLog
```

```
    self storeObject: (self convertLog: aLog)

FormattingLogger>>storeObject: anObject
    "Hook to customize to specify how an object should be stored once converted."

    self subclassResponsibility

FormattingLogger>>convertLog: aLog
    ^ self formatter
        formatLog: aLog
        from: self
```

Note that this design decomposes the actual formatting from the storing: it is possible to then store on different media the formatted version of our log objects. StringStreamLogger will store formatted logs in a stream but other subclasses may store the items into a database.

A default formatter is created when none if specified. It is an instance of LogFormatter. LogFormatter is a simple class that defines two methods: formatLog:from: and formatLog:on:from:.The first one just returns a formatted string for a log object, while the second one pushes a formatted string to the argument stream.

```
LogFormatter>>formatLog: aLog from: aLogger

    ^ aLog printString
```

```
LogFormatter>>formatLog: aLog on: aStream from: aLogger

    aStream cr.
    aLog printOn: aStream.
```

Now we are ready to define the behavior of StringStreamLogger. We redefine the storeObject: method to write to a stream that by default is defined to return Transcript.

```
StringStreamLogger>>storeObject: aLog

    self stream cr.
    self stream << (formatter formatLog: aLog from: self)
```

```
StringStreamLogger class>>defaultStream
    ^ Transcript
```

1.5 Conclusion

This simple frameworks is the foundation to get modern object-oriented logs and it opens a complete space to future ideas.