Logging guidelines
(for you and your mates)

August 18th, 2011

Always dreamt about discovering *the 4 questions a developer should answer before writing a new log entry*? This document is for you!

You've got a train to catch? directly read the green frames all along the document, and the famous *4 questions...* at the end. Have fun!

Version	**2.7**
Status	
Author(s)	**Thomas PIERRAIN**
Submission Date	**August 19, 2011**
Reviewer(s)	
Review Date	
Approved by	
Distribution List	**DEV but also ARC and PRD teams**

Modification & Change

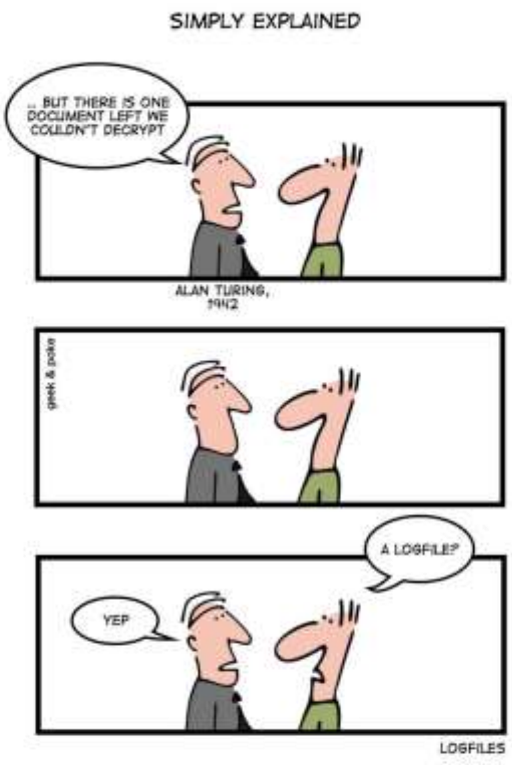
Version	Date	Cross Reference	Modified by	Description
1.0	04/12/08		Thomas PIERRAIN	First version
1.1	04/12/08		Thomas PIERRAIN	Adds the technical or functional log entry type
1.2	09/12/08		Thomas PIERRAIN	Adds few changes (introduces static class for logger names, changes tokens, changes log format part, adds recommendation for AsynchronousAppender, adds a new "question for developer", etc)
1.3	09/12/08		Thomas PIERRAIN	Modifies few details + check spelling
1.4	17/12/08		Thomas PIERRAIN	Completes the log format part
1.5	22/12/08		Thomas PIERRAIN	Changes the performance impacts sample and sets the correct tokens within the Technical or functional sample
2.0	18/07/11		Thomas PIERRAIN	Removes references to ITEC/FIC/STT. Replaces the technical or functional tokens by log4{x} levels. Introduces a new "How many file?" section.
2.1	19/07/11		Thomas PIERRAIN	Changes paragraphs order, Adds the *"it's all about GREP -ability!"* and *"Smoke test your logs before going to production"* parts, Adds log4net default configuration sample in appendix, Adds Apendix for Iaplus.Logging, Follows Spolsky advices.
2.2	20/07/11		Thomas PIERRAIN	Fixes some spelling issues and the log4net configuration sample.
2.3	21/07/11		Thomas PIERRAIN	Puts emphasis on some points (with green frames), Downgrades the EXCPT level
2.4	25/07/11		Thomas PIERRAIN	Introduces Dog tags for log files, sets UTC timestamps within the the log4net default configuration, Adds a FAQ in the appendix.
2.5	26/07/11		Thomas PIERRAIN	Adds the "Within the same standard location" chapter, Rewords some parts
2.6	18/08/11		Thomas PIERRAIN	Introduces the logging guidelines poster in appendix, adds precisions regarding the performances, adds a comic strip (thanks to Pierre-Yves AIMON) within the introduction.
2.7	19/08/11		Thomas PIERRAIN	Mitigates the DRY principle.

References

Ref	Title	Date	Version	Author(s)

1 Introduction

The trigger for this document is a general admission of failure: **our applications are too often hard to support due to scrappy logs.** Our logs come with too much information, not very readable nor understandable, with too much technical stuffs or even false alerts[1], etc.



We –as developers- need to supply more useful and relevant informations to our support and QA teams. **This document aims to help every developer to log in a unified and useful manner**.

This document was establish long time ago, after several brainstormings and reviews with Java and .NET developers (Cedric MOTSCH, Julien AYME, Prosper EGUE, Fabien BONDI, Julien ADAM, Philippe PEURET, Rawad KHODR, Thomas ALIROL, Cyrille DUPUYDAUBY, François DUMONT), QA and Support team members (respectively James WALL and Pascal PETIT).

Some recommandations rely on the de-facto standard log4{x}[2] frameworks. Others are more generalists.

Anyway. Since we don't want to play like the *lazy bast… developers* that used to start coding without a use case, it's time to take a tour of our needs first, before making some recos.

[1] Who never saw some "ERROR" or "WARN" log entries saying that everything was going fine ;-(

[2] log4j or log4net

2 Logging for what?

The purpose of logging is to keep track of the history of a process execution in order to understand and explain what happened in our systems afterward. This can be required:

- When a problem occurred (unexpected behavior of our system) – let's call those log entries "standard log entries"
- For audit purpose (legal constraints) – let's call those log entries "audit log entries"

3 Logging for whom?

You can't have your cake and eat it too! Unless if you want to get dumped by your girlfriend (because she never saw you before 10 PM last months) or if you liked to be insulted by angry users, your application logs should target the support teams first (and not you). Thus, **every "standard log entry" should be designed to help our support team members**:

- to be able **to handle a problem autonomously** (otherwise how to code properly when the phone is constantly ringing…)
- **to be more accurate when they forward the context of a problem to the level 2 support teams** (usually us-the dev teams-and usually with a lack of relevant information from our point of view)
- **to correlate events from several applications** that are related (following a functional track like an RFQ[3], an order, a trade, etc)

LOGS ARE FOR A SUPPORT AUDIENCE!

And if you don't have empathy towards your support team buddies (which by the way, is a shame), you just have to imagine that YOU will be THE level-2 support guy for the application-you-never-heard-about-before-last-week, during the Thailand holidays (2-weeks) of one of your other dev mate[4] ... You've got it?

Relevant application logs should save us lots of time, stress and energy. Collectively.

So let's do it, and let's get started with the recos now!

4 First: it's all about GREP[5]-ability!

Make your log entries grep-able. Pick what is about your system, and **provide markers for your audience**.

[3] Request For Quote

[4] True story ;-(

[5] grep is the famous command-line text-search utility originally written for Unix

The choice of proper grep markers is THE key-success for the relevance of your application logs.

You develop a MarketMaking contribution engine? Pick the publication topics (with instruments, products or rate curves ids) as markers. You develop a Quotation Request Manager? Pick the RFQ and customer or customers groups ids as markers.

You develop a post-trade system? Pick a unique trade id as marker for your log entries. To drive the redaction of your log entries, ask yourself: "what kind of requests angry users will make to my support team buddy?".

★ **USE LOG PATTERNS THAT CAN BE EASILY RECOGNIZED BY 1st GRADE STUDENT'S REGULAR EXPRESSIONS**

One simple grep instruction should allow the support team operator to retrieve the entire story of a trade, an RFQ or a customer interaction.

5 Followed by a correct usage of the Logger name

The first and foremost advantage of any logging API over plain System.Console.WriteLine(…) resides in its ability to disable certain log statements while allowing others to print unhindered. This capability assumes that the logging space, that is, the space of all possible logging statements, is categorized according to some developer-chosen criteria.

Logging requests are made by invoking one of the printing methods of a logger instance (through the log4net.ILog for instance). Both the de facto-standard log4J and log4net logging APIs provide hierarchical naming rule for loggers.

Loggers are **named entities**. Logger names are **case-sensitive** and they follow the **hierarchical naming** rule:

> *Named Hierarchy*
>
> *A logger is said to be an ancestor of another logger if its name followed by a dot is a prefix of the descendant logger name. A logger is said to be a parent of a child logger if there are no ancestors between itself and the descendant logger. The hierarchy works very much in the same way as the namespace and class hierarchy in Java or .NET. This is very convenient.*
>
> *(source: apache log4net manual)*

The first important choice you have to make about logging is to decide on a scheme that assigns each log message to a particular category.

Unfortunately, it is common practice **to use the fully qualified name of each class whose activity is being logged as a message category** (~~because this allows developers to fine-tune log settings per class?~~ because of laziness!). This, however, **is not a "support-oriented" way of logging** within an application **and should be avoid**.

★ **DON'T ACT LIKE A NERD! (by using class names as logger names ;-(**

Imagine one sec the support team member trying to figure the name of the logger he must enable under production (and stress) to better understand what happens... it's doesn't make sense! As a consequence, DO NOT code something like:

```
private static ILog logger = LogManager.GetLogger(typeof(this));
```

Need another motivation? In an application, there are many other kinds of log messages. For instance, one log message might be produced for security advisors, while another could be produced to aid performance tuning. **If both messages concern the same class and are thus assigned the same category, it will be difficult to distinguish them in the log output**.

Ok.To avoid this problem, our applications should have a set of specialized "root logger" [6] names with distinct categories:

- Security
- Business
- Referential
- Persistence
- Configuration
- Communication.Input
- Communication.Input.Requests
- Communication.Output
- Communication.Output.Replies
- Monitoring …

Each of these "root loggers" can be configured with its own priority and output handler. For example, the security logger can encrypt a message before it is written to the destination, etc. Also:

★ **NEVER DROP PASSWORDS IN YOUR LOGS.**
OTHERWISE YOU WILL BE DECAPITED WITH A GARLIC PRESS!

To prevent from logger name case sensitivity or misspelling issues, it is highly recommended to pick/build the logger names through a dedicated static class with root logger names as const properties.

[6] « root loggers » in the hierarchical naming rule way (meaning: prefixes for every logger of the application)

6 With also a correct choice of logging level

Messages that fall within a single category (security, say) can have different priorities. Some messages are produced for debugging, some for warnings, and some for errors. The different priorities of messages are represented by logging levels.

The basic logging levels in log4{x} are:

- DEBUG
- INFO
- WARN
- ERROR
- FATAL

Both the de-facto standards log4j and log4net APIs provide logging levels beyond these basics. The primary purpose of a logging level is to help you filter useful information out of the noise.

Technical, functional?

It is very useful for support teams members to be able to “grep” only the functional log entries of an application (removing the technical related noises).

Since we need a token to be able to grep properly, it appears useful to benefit from the log4{x} extension mechanisms by introducing new logging levels: **FUNCT**, **TECH**

For instance, a simple log fragment like this[7]:

```
[FUNCT]  Requests credit check for customer 'john Doe'. (Business.CreditCheck)
[TECH]   Encrypts the credit check request message.  (Security.Encryption.CreditCheck)
[TECH]   Sends the credit check request to the service.  (Communication.Output.CreditCheck)
[TECH]   Receives credit check answer for the customer ‘john Doe’ (Communication.Input.CreditCheck)
[TECH]   Decrypts the credit checkdata received from the server.  (Security.Encryption.CreditCheck)
[FUNCT]  Credit check confirmed for the customer 'john Doe'. (Business.CreditCheck)
```

May be grep-ed to output only the functional log entries:

```
[FUNCT]  Requests credit check for customer 'john Doe'. (Business.CreditCheck)
[FUNCT]  Credit check confirmed for the customer 'john Doe'. (Business.CreditCheck)
```

Exceptions and Traces

It may also be useful to introduce a new logging levels related to exceptions we are throwing (EXCPT). In some (very rare) cases, that would be very handy to log every exception we throw within our systems. Since exception may be hidden “under the carpet” by code, it is nice to have an ultimate way to check whether a given exceptions have been thrown or not. **On the other hand, we don’t want to log twice a given Exception by default** (we may also be decapited for that…). As a consequence, this new EXCPT level should be placed between the TRACE and the DEBUG levels. Oh yes, because the existing but somehow hidden TRACE logging level of the log4[x} frameworks should be exposed and used.

At the end of the day, we should be able to configure and to use the following log4{x} logging levels:

- TRACE

[7] Note here, that I removed the timestamps within log entries in order to gain space and to focus only on what matters in that paragraph.

- EXCPT
- DEBUG
- TECH
- INFO
- FUNCT
- WARN
- ERROR
- FATAL

To avoid using the wrong level and thus reducing the usefulness of log messages, developers must be given clear guidelines before they start coding.

- **TRACE**: Messages in this level are used to fully check / follow the code path execution of a program

- **EXCPT**: Messages in this level contain explanations related to Exception **throw witin our code** (they are mostly used for rare problems diagnosis).

- **DEBUG**: Messages in this level contain extensive contextual information we don't normally want to log in production, but might want to log if there are problems (they are mostly used for problem diagnosis).

- **TECH**: Messages in this level might help the support to follow/understand what happened in detail in our system (every technical related steps of our programs execution).

- **INFO**: Messages in this level might help the support to follow/understand what happened in our system (the major steps of our programs execution). Usually information about starting and stopping state, any data which we want to log in production

- **FUNCT**: Messages in this level might help the support to follow/understand what happened in our system from a functional point of view (eg:. following an RFQ, a deal…)

WARN: A warning message indicates a potential problem in the system. **Every WARN message should trigger a validation/corrective action by the support team on a periodical basis** (every evening, every end of week, every end of two weeks)

ERROR: An error message indicates a serious problem in the system. **Every ERROR message should trigger a validation/action/manual intervention by the support team ASAP!**

FATAL: Messages in this level indicate that the application is expecting to shutdown/suicide. **If the application is still up after a FATAL log entry, the support team should help the application to get stopped / killed ASAP**.

DON'T *FALSE ALERT* YOUR MATES!

Since every WARN or ERROR log entry should trigger an action by the support team, it is very important to ensure that we choose the correct log level when we log within our applications.

PTFM[8]

If you want to get even nicer and efficient with your related support team buddies, **it is also a good idea to give them the list of possible [ERROR] or [WARN] messages** (in excel for instance), **with a brief line on what each of them mean**.

7 Split into several files

Again. Don't forget that logs are for a support audience. Everything that could decrease the number of steps a support team member should make in order to get the needed information should be done.

Thus, **it is recommended for an application to produce only one type of log file** (instead of having several log files to be correlated: one log file for a given topic of the application, another log file for another topic, etc).

Since support team members have tools allowing them to easily "grep" from a range of files, **it is also recommended to choose a rolling file strategy** (see the log4{x} RollingFileAppender documentation or the configuration sample provided in the Appendix of this document) that will split your application log file into several parts (let's say **with a max file size of 100Mb for instance**). This will allow those tools **to parallelize/speed up their "grep" operations**.

EVERY LOG FILE, SHOULD HAVE ITS DOG TAG!

Because logs are moving around (between teams or servers), their file names should include the name of the application instance, the identifier of the machine (netbios or ip) and the date (yyyy-MM-dd}.

See a log4net RollingFileAppender configuration sample in the Appendix.

Eg.: MyApplicationName(SRVPARPOKFXP05)-2011-07-25.log

8 Stored within the same standardized location

Don't act like Dogbert[9]! Support team members have several tools/alerts/shortcuts to configure per application. Be nice, and adopt their norms by configuring your logs to output always in the same standardized directory: **\\<serverName>\homeware$\Logs\<ApplicationName>\<InstanceName>**[10]

[8] PTFM stands here for: **P**roduce **T**he **F**… **M**anual

[9] the evil network systems admin: http://dilbert.com/strips/comic/1997-08-28/

[10] For windows servers

9 Configured with a proper message format

Each line of log must contain at least the following information:

- UTC Hours (24 hour basis)
- UTC Minutes
- UTC Seconds
- UTC Milliseconds
- Level (enclosed with brackets[])
- Message (**no carriage return allowed!**)
- Logger name
- Thread name or identifier

Eg:

```
18:23:25,766 [ERROR] The RMDS infrastructure for the managed publication for source name 'POK_DEV6' did not respond properly  [Communication.Output.Rmds (4582)]
```

★ **USE UTC TIMESTAMPS!**

Coordinated Universal Time is a must when you'll have to correlate and to work with log files produced on separate servers. You won't regret it![11]

WHEN POSSIBLE, DON'T REPEAT YOURSELF (DRY)

Because we already put the date within the log dog tag (ie. its name), it is useless to remind the date for each log entry.

"When possible"?!? Let me explain: if you are using a logging framework that prevent you from being able to use a composite rolling file strategy (meaning: with the ability to roll log files on a date period AND within a date period on file size), you should probably abandon the DRY principle to ease your logs exploitation[12].

Too bad for the wasted disk spaces and the grep performance impacts.

The DRY principle reminded: "*every piece of knowledge must have a single, unambiguous, authoritative representation within a system*" (source: The Pragmatic Programmer[13])

[11] But also check whether your servers clocks are properly synchronized or not.

[12] Because you don't want to get confused by distinct days log entries within the same log file, without knowing it.

[13] The Pragmatic Programmer, from journeyman to master (Andrew Hunt, David Thomas) - 2000

10 With the least impact on performances

★ **LOG ASYNCHRONOUSLY, BUT NEVER CRASH BECAUSE OF LOGS…**

Otherwise you will put shame on your 3 future generations…

To avoid that, you must **use an asynchronous appender** (see Iaplus.Tools for .NET applications) to prevent from introducing latency within your code due to log activity. While it's a shame that the log4net library doesn't provide such a requested appender, you could hopefully use the SG CIB Iaplus.Logging AsynchronousAppender (available in 2 parts[14] under the anthology component catalog: http://anthology.fr.world.socgen/Search.aspx?PageMethod=SimpleSearch&query=ia%252b.logging).

You also must **set a proper threshold value for the AsynchronousAppender buffer to prevent your process from having OutOfMemoryException due to log congestion** (since the AsynchronousAppender is usually configured with a BelowNormal (thread) priority, you may encounter situations where it won't have enough time slice to dequeue log events when your application will be under massive load).

Also, **avoid to consume CPU in building string messages for log entries that finally won't be appended** (depending on your log4{x} configuration settings).

Thus, it is required to test the Is…Enabled boolean properties/methods before paying the cost of a message string construction.

```
if (logger.IsDebugEnabled)
{
      Logger.Debug ("Start the strategy of the pricing engine.");
}
```

Or even better,

★ **USE THE …Format(…,…,…) OVERLOADS OF THE ILog INTERFACE**

like:

```
Logger.FunctionalFormat("Requests credit check for customer '{0} {1}'.", customer.FirstName, customer.LastName);
```

which internally make the Is…Enabled check for you (increasing your code readability with neglectable performance impacts[15]).

Note: If the arguments passed to the …Format involve CPU or time consuming properties (or methods) calls, you'd better keep your old Is…Enabled strategy.

Some other performance optimizations might also be achieved in some critical systems, to reduce the impact of the log activity on processes that rely on Garbage Collector[16] (like in .NET or Java). The

[14] Iaplus.Logging.Interfaces.dll and Iaplus.Logging.dll

[15] When Debug is not enabled, you will only save less than 10 nanoseconds by using a simple IsDebugEnabled statement instead of a call to the DebugFormat(…) overload.

general idea would be to prevent from increasing the memory pressure by forwarding the log construction and append task to another process that won't bother to have GC freezes (like Gen2 collection in .NET for instance).

But I won't detail this here (it may probably worth another paper).

11 Last but not least: don't forget to smoke test your logs before going to production

Before the end of your iteration (and a possible release under production), you should absolutely run your application with the default logging configuration[17] to see whether your logs are accurate or not.

★

ITERATIVELY:
RUN YOUR APP, CHECK YOUR LOGS, AND FIX THEM UNTIL THEY ARE FINE

They should be easy to read, but also easy to grep. While your are testing your favorite grep instructions, you should embed them into your support documentation/release note to help your support team buddies.

Hunt the false alerts![18] make your logs a tool to earn time, efficiency and to reduce stress in the future.

[16] That may freeze you in situations where you don't allocate memory accurately

[17] with INFO as the root default logging level.

[18] I even knew a tough guy that may prevent applications to go in production if the QA team founds some ERROR within its logs.

12 Appendix

12.1 The 4 questions a developer should answer before writing a new log entry

Pin those 4 questions on the wall in front of you until you don't need it ;-)

1. What logger name should I use? What "root logger" name should I use as prefix for this logger name?
2. What logging level should I use?
   a. Does this new log entry should trigger an immediate action from the support team? (=> ERROR level)
   b. Does this new log entry should trigger an afterward action from the support team? (=> WARN level)
   c. Does this new log entry indicate a functional step of the system? (=> FUNCT level)
   d. Does this new log entry indicate a strong step / action of the system? (=> INFO level)
   e. Does this new log entry should inform that an exception is instantiated and thrown? (=> EXCPT level)
3. Does this log entry might occurred frequently? (=> if yes, I must use the IsDebugEnabled or the proper Is…Enabled property/method before building this log entry string message)
4. **Will this log message be readable / grep-able by the support team?**

12.2 Sample of log4net default configuration

```
<?xml version="1.0" encoding="utf-8" ?>
<log4net>
  <appender name="FileAppender"
            type="log4net.Appender.RollingFileAppender">
<file type="log4net.Util.PatternString" value="MyApplicationName(%property{log4net:HostName})-
%date{yyyy-MM-dd}.log" />
    <!-- do not specify minimal locking model it could be terrific for performance -->
    <datePattern value="yyyyMMdd" />
    <appendToFile value="true" />
    <staticLogFileName value="true" />
    <maxSizeRollBackups value="-1" />
    <countDirection value="1" />
    <rollingStyle value="Composite" />
    <maximumFileSize value="100MB" />
    <layout type="log4net.Layout.PatternLayout">
      <conversionPattern value="%utcdate{HH:mm:ss,fff} [%-5level] %message  [%logger
(%thread)]%newline"/>
    </layout>
  </appender>
  <appender name="AsyncAppender"
            type="Iaplus.Logging.AsyncAppender, Iaplus.Logging">
    <param name="Priority" value="BelowNormal"/>
    <appender-ref ref="FileAppender"/>
    <param name="StopEnqueuingLogEventsThreshold" value="70000"/>
    <param name="ResumeEnqueuingLogEventsThreshold" value="45000"/>
  </appender>
  <root>
    <level value="INFO"/>
    <appender-ref ref="AsyncAppender" />
  </root>
</log4net>
```

12.3 The logging guidelines poster ★

LOGS ARE FOR A SUPPORT AUDIENCE!

USE LOG PATTERNS THAT CAN BE EASILY RECOGNIZED BY
1st GRADE STUDENT'S REGULAR EXPRESSIONS

DON'T ACT LIKE A NERD! (by using class names as logger names ;-(

NEVER DROP PASSWORDS IN YOUR LOGS.
OTHERWISE YOU WILL BE DECAPITED WITH A GARLIC PRESS!

DON'T *FALSE ALERT* YOUR MATES!

EVERY LOG FILE, SHOULD HAVE ITS DOG TAG!

USE UTC TIMESTAMPS!

WHEN POSSIBLE, DON'T REPEAT YOURSELF (DRY)

LOG ASYNCHRONOUSLY, BUT NEVER CRASH BECAUSE OF LOGS…

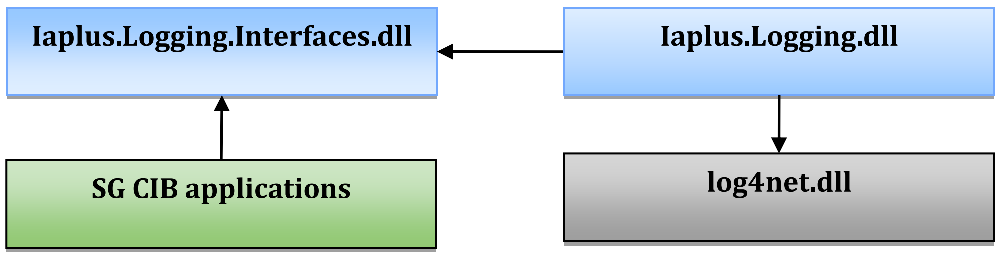
USE THE …Format(…,…,…) OVERLOADS OF THE ILog INTERFACE

ITERATIVELY:
RUN YOUR APP, CHECK YOUR LOGS, AND FIX THEM UNTIL THEY ARE FINE

12.4 Few words about the Iaplus.Logging .NET library

The .NET **Iaplus.Logging** library **has been built** within SG CIB **to encompass most of the logging guidelines presented within this document**. This library is IoC compliant, wraps the unique log4net assembly[19] and consumes the standard log4net configuration file.

As indicated within the dependency diagram below, the Iaplus.Logging.Interfaces.dll does not reference the log4net assembly:



This library also provides an Asynchronous Appender[20] providing a safe-threshold mechanism in order to prevent from having OutOfMemoryException when CPU is no more available for the AsyncAppender. The AsyncAppender should be configured (see the log4net configuration file presented above) with a threashold level (max amount of log entry to buffer). If the AsyncAppender reached this max amount of log entry to append[21], it should not accept more log entry anymore and log only one message indicating that it won't log anymore since the other threashold[22] has been reached.

Iaplus.Logging is quite **simple to use for log4net users**. It provides 4 major log4net-like interfaces:

- **ILogger** (to support the logging for levels Debug, Info, Warn, Error and Fatal)
- **IExtendedLogger** (which extends ILogger, but also to support the logging for levels Trace, Functional and Technical; some Debug overloads are also provided for log entries related to Exception throwing)
- **ILoggingFactory** (to get loggers instances given their names)
- **ILoggingEnvironment** (to configure the logging environment and to provide the ILoggingFactory)

[19] By the way, with all its interface types, it is quite strange that the log4net library was not shipped in two assemblies (one for the interfaces, the other for the implementations).

[20] a re-brand of the one that was embedded within the deprecated Iaplus.Tools library

[21] defined through the "StopEnqueuingLogEventsThreshold" AsyncAppender property

[22] defined through the "ResumeEnqueuingLogEventsThreshold" AsyncAppender property

12.5 FAQ

Ok. These are the questions you probably asked yourself while reading this document. If I missed some other points, don't hesitate to help me to increase this FAQ (thomas.pierrain@sgcib.com):

1. **Since 'Functional' or 'Technical' are completely orthogonals to the criticity notion, why do you finally recommand to use new log4{x} levels to handle them? Are you inconsistent or tired?**

   A No I'm not. Thanks.

   A We initially thought that it would be a good idea to introduce new tokens to distinct technical (token: [.t]) or functional (token: [.f]) stuffs. The original idea was to build a wrapper upon log4{x}[23] to automatically add those tokens within our log entries depending on the method overloads used by the developer (eg:. Logger.InfoTechnical(..) or Logger.InfoFunctional(…)).

   In fact, we made a slighty mistake at the time: **while it can be very useful to grep only the functional log entries of an application**, the extraction of the technical log entries only is usually not very accurate (**technical logs are rarely relevant without more functionnal context**).

   Furthermore, it's seemed to us that to rely on the 'LEVEL' extensibility mechanism to fulfill this use case was more accurate and close to the initial spirit of the log4{x} APIs.

2. **Why shouldn't we put the date within each log line?**

   A Not only to respect the DRY principle, but mainly because our default production servers still have tiny hard drives ;-(

   But if you need it[24] and you can afford it, be my guest!

3. **Instead of temporarily stopping the acceptance of new log entries when the AsyncAppender is no more able to dequeue, why don't you choose a strategy based on levels criticity in order to filter what's important to append and what's not in that crisis situation instead?**

   A It's an endless debate, but here are several elements to explain our default AsyncAppender threshold strategy. **Firstly**: it is crucial not to crash because of logs[25], the strategy must be robust enough to avoid ANY kind of situations that might lead to that shame. Temporarily stopping to accept new log entries seems the most reliable way to achieve that. Furthermore, when a server is no more able to Append its own logs. It is highly probable that it will trigger more unexpected ERROR log

[23] We finally did it. See the Iaplus.Logging library laius in the Appendix

[24] For instance if your logging framework is not providing composite rolling file strategies (unlike log4j or log4net). See previous explanations within the paragraph "Configured with a proper message format"

[25] Due to OutOfMemoryExceptions.

entries (that you won't be able to cap). **Secondly**: **it is more interesting to have the logs that show how a system got crazy, rather than to have the logs of a crazy system**. **Thirdly**: any attempts to provide a strategy based on max number of log entries to accept per level (INFO, ERROR,..) would lead us to more confusion (with missing log entries per level depending on their levels. As a consequence, nobody will never trust your logs again. Return to square one ;-(