What's the point of a logging facade? [closed]

Asked 14 years, 2 months ago Modified 5 years ago Viewed 20k times

50

Closed. This question is opinion-based. It is not currently accepting answers.

Want to improve this question? Update the question so it can be answered with facts and citations by editing this post.

Closed 6 years ago.

Improve this question

There are a bunch of different logging libraries to choose from, each with their own set of quirks and advantages. (.NET examples: log4net, System.Diagnostics.TraceSource, NLog, etc.)

The natural inclination is to abstract away those quirks and use a logging facade. (examples: Castle.Services.Logging, Common.Logging, Simple Logging Facade) That way, if a given logging framework that you're using goes stale, or a different one comes into vogue, you can just swap out the implementation and leave the code untouched.

But there are multiple logging facades to choose from. Given that the answer to many disparate logging implementations was abstraction, why not use a logging facade facade? If that sounds ridiculous, what makes it more ridiculous than the original logging facade? What makes one extra layer of abstraction on top of the logging framework the magic number?

design-patterns **logging**

Share

Improve this question

Follow

edited Nov 29, 2019 at 15:16

Luke Girvin

13.4k ● 10 ● 67 ● 85

asked Sep 29, 2010 at 19:46

Brian

5,956 ● 12 ● 61 ● 82

12 joelonsoftware.com/articles/fog0000000018.html – Nick Van Brunt Sep 29, 2010 at 20:18

@brian I'm not sure your assumption about people's natural inclination is true. Insist most developers don't even think about it. – Howiecamp Jan 6, 2016 at 19:35

9 Answers

Sorted by: Highest score (default)

I will speak mainly from the perspective of using the abstraction to insulate application code from a particular logging framework. There are other factors that can affect

67

one's choice of logging framework or one's choice of (and requirements for) an abstraction.

I have spent a lot of time recently evaluating various logging frameworks as well as third party logging abstractions.

Some people feel there is value in insulating their application code from a specific logging framework. You will find many posts here on SO like this and this and this(and there are more) where logging is discussed and many people take it as a matter of course that the logging framework should be wrapped/abstracted.

Obviously, this allows you to not be tied to a specific framework. Is this important? Will you ever really switch out your logging framework? Well, there are also plenty of people who either don't mention wrapping or those who recommend against it. If you look at some of the examples of logging framework wrapping code that has been posted here, you can also see many examples of why at least some people should not wrap their logging framework!

If you had started a project recently, you might have examined logging frameworks and, perhaps, narrowed it down to two finalists: log4net and NLog. Each has arguments in its favor. log4net is clearly a favorite, probably THE favorite of those who have expressed an opinion. NLog provides very similar capabilities. Judged by popularity, log4net might be the clear choice. Based on capabilities, they seem very similar. Based on "recent

activity" (as indicated by checkins to their source code repostories by blog activity or lack thereor), NLog be the clear choice. If you had to pick a year ago, you might go with log4net since it would be the "safe" choice. It was not clear when NLog would release. In the year since, NLog has gone through a pretty signifcant development cycle, releasing a beta version just a few days ago.

Which to choose a year ago? Which to choose now? Was one a clearly better choice then? Is one the better choice now?

One thing an abstraction gets you is the ability to put off the decision of which one to choose (you don't necessarily even HAVE to choose EVER, although you probably want to if you plan to deliver the logging framework with your product). You can test drive one and then the other and get a feel for how they work with your application, with your team, in your environment. Using something like Common.Logging or SLF allows you start writing code now, coding to some logging interface/API, and getting your logging code in place. If you believe that the interface/API is provided by the abstraction is sufficient for your work (and, why wouldn't it be since it is essentially the same as the interface/API provided by log4net and NLog), then there is not much danger in using the abstraction. As you go through the development cycle, you might find that one framework or the other better suits your needs. Having coded to the abstraction, you are free to make that choice at any point, up until the time your product goes out the door.

You might even be thinking, in the back of your mind, that you could possibly write a logging library from scratch. Again, if you believe that the interface/API of log4net and/or NLog is sufficient, you might implement your logging library with a similar API. If you believe that, that might be another reason to use an abstraction. Again, you can start writing code (for your product, not your logging library) today, logging with some other logging framework until such time that your "from scratch" logging library is ready. Maybe you really want to use System.Diagnostics.TraceSource and Ukadc.Diagnostics (to get output formatting capabilities similar to log4net or NLog) so that you can get "better" integration with the logging that Microsoft has implemented in some of their platforms using TraceSources. It could be pretty easy to write a "logger" in terms of TraceSources and then write the abstraction so that you could plug it into Common.Logging or SLF. (If the interface/API is sufficient, you could just write your "logger" in terms of the abstraction library's interface and not have to write an additional abstraction layer).

With such persuasive arguments as these, why would anyone ever NOT use an abstraction? Ha ha, just kidding!

If an abstraction is good, should you write your own or use an existing one? If you write one on your own, then you obviously have to write it. How does one do this? Well, you might just define an interface and wrap one framework (be careful and wrap it correctly!). Later, if you

decide you want to switch, wrap that framework. If you are careful, you don't have to change any application code, except for maybe the place where you actually create the underlying framework's objects. Maybe this is good. You have avoided a dependency on some third party abstraction for the "small" price of implementing a single wrapper over a single framework. However, there is a cost. Until you have written your abstraction you cannot really write a lot of application code that has logging in it, unless you have a good strategy for changing it over to your abstraction. It also becomes more difficult to test drive two or more frameworks to decide which works better for your. Each framework that you want to "try" requires another wrap job. If you want to switch among frameworks easily (at least during development cycle), you have work to do to make it easy. The third party frameworks provide this out of the box.

Wow! Now I'm sold! Give me logging abstraction, or give me death!

Are logging abstractions all gravy? Is there a downside? They can't THAT great, can they?

Well, as always, when "buying" something or when getting something free, you get what is available. Logging abstractions are no different. Neither Common.Logging nor SLF expose at least one very important set of capabilities of log4net/NLog - the logging context capabilities (GDC, MDC, NDC). These can be key to getting adequate information logged and formatted to

enable you to get the most value from your. SLF does not provide a TraceSource abstraction. It also does not provide IsXXXEnabled functions. Common.Logging provides a TraceSource abstraction. Castle.Logging DOES expose GDC/MDC/NDC for log4net and NLog. It also provides a TraceSource abstraction. Castle's TraceSource abstraction also enhances TraceSource logging by providing a "hierarchical" naming capability, similar to that provided by log4net and NLog. It looks pretty cool!

Also, these projects are all opensource of one form or another. So, depending on the abstraction, the developers might have more or less of a vested interest in keeping it up to date and adding new features. Common.Logging has been through a few versions and is used, AFAIK, in Spring.Net. Seems reasonbly active, at least historically. Castle.Logging is used in the Castle framework. So, they apparently have "real" customers and are getting "real world" usage, which will hopefully drive more feature implementation. SLF, as far as I can tell, is not used as part of a "real" development platform, so it is hard to tell how much it is exercised.

It is not clear what the roadmap is for these platforms. Common.Logging has some upcoming features listed on their website, but not clear indication when they will be available. The website says "June", but of what year? How often is the mailing list monitored? For SLF, how often is their codeplex monitored? Where does the priority of these "free" projects rate compared to the

developers' paying jobs? Can you afford for some third party abstraction to implement a feature that you need? Will they be receptive if you implement something and then submit it back for consideration to be included in the product?

On the plus side, all of the source for all of these abstractions is available, so you could just assume responsibility for it and make any fixes or add any enhancements that you which, without having to go through time and energy of creating an abstraction from scratch. Do you like Common.Logging but really want log4net/NLog GDC/MDC/NDC? Get Castle's implementation and add it to Common.Logging. Voila! A logging abstraction that contains nearly 100% of the log4net/NLog logging API. Do you prefer SLF but wish it had IsXXXEnabled? Not much work to implement that. Go ahead and tack on the GDC/MDC/NDC while you are at it. Do you like Castle? (I'm not that familiar with it, not sure how easy it is to use outside of Castle, if that matters) Be careful, I haven't used it, but looking at the source on git, it looks like the NLog logger abstraction *might not* retain call site info.

Is it ethical to take parts of multiple open source projects and combining them to make one "super" project (for your own or your company's use)? Is it bad to take Common.Logging and augment it with Castle's GDC/MDC/NDC implementation? I don't know. I'll let someone else answer that.

I'm nearly finished...

Some third party logging abstractions provide other capabilities. You might use a library that is implemented in terms of, say log4net. You might not want to use log4net, or at least might not want to be tied to it. Common.Logging (and maybe SLF) makes it relatively easy for you to capture the log4net logging messages and reroute them through the abstraction so they are captured in the abstraction's underlying logging framework's logging stream. SLF might provide something similar. Of course, you might be able to do something similar with existing logging frameworks, either out of the box or by writing a custom log4net Appender, NLog Target, or System.Diagnostics TraceListener. These features have not bubbled up very high in my particular evaluation of whether or not to use a third party logging abstraction on my project because I am mainly interested simply in the abstraction aspect.

So, where do I stand? I think that there is value in keeping your application code insulated from a specific logging framework. To me, Common.Logging looks like a solid choice of abstraction, although some important features are missing (GDC/MDC/NDC) and it is not Silverlight compatible. It would great of those features became available soon. I am comfortable with implementing GDC/MDC/NDC if I have to. Making it Silverlight compatible would probably take more effort, primarily because I am not particularly experienced with C#/.NET/Silverlight. Until those issues are ironed out, we

would be able to write plenty of application code with Common.Logging in place. We can spend our time developing our application rather than developing yet another logging library or abstraction library. If we end up having to add those missing features ourselves, well, we would have had to do a lot of that if we had implementing a logging library or abstraction library ourselves.

Share Improve this answer

Follow

edited May 23, 2017 at 12:18

Community Bot

1 • 1

answered Sep 30, 2010 at 4:36

wageoghe

27.6k • 13 • 89 • 117

3 Such a massive post! I agree with everything, +1 there certainly, but perhaps it could be simplified/formatted a bit? I can see lots of things stated multiple times for instance. Also, since this is very old, do you have any newer insight on this? Has something changed in the meantime that would warrant discussing again? – julealgon Feb 21, 2014 at 14:46

1 I don't have anything to add since my original post. At the time I wrote this, I was in the throes of learning as much about logging as I could. In the case of my organization, we settled on using the logging facade approach, specifically Common.Logging for .Net. We also settled on NLog as our underlying logging system, although, due to the facade, we are not directly dependent on NLog. – wageoghe Mar 7, 2014 at 17:22

FWIW: I recently stumbled on a LibLog which is just a file and not a dependency. – LosManos Apr 15, 2015 at 7:07

9

I think what makes One (level of abstraction) the magic number here is that Zero is too few and Two is too many.

Swapping a logger behind a logger facade (number of levels: 1) can possibly result in some user benefit, such as the new logger can do something that the old logger can't. I can imagine that it could be performance, supporting certain types of appenders, etc.

It's much harder to imagine the user benefit from swapping a logger facade (number of levels: 2).

(And if the number of levels is 0, then it's probably just bad object-oriented design: you'll have thousands of places in your code where the logger is referenced and what if there's a breaking change in the next version of the logger.)

The deal with logger facades appears to be that you have to pick one of the third-party options or to create your own and prepare to stick with it for a long time.

Share Improve this answer

Follow

answered Sep 29, 2010 at 20:40

azheglov

5,513 ● 1 ● 23 ● 30

5

An important use of a logging facade is when you're **writing a library**. Embedding dependencies in a library is always something that requires a bit of care, and logging even more so.

In short, **you don't want to force your logging implementation onto your library users**. Using a well-chosen facade means they'll be able to handle your library's logs with their own framework of choice and won't have to go through some weird dependency exclusions and loopholes to make both your logging framework and theirs coexist.

Share Improve this answer Follow

answered Aug 23, 2017 at 17:38

user136698

3

Until NLog and log4net provide an interface that can be used instead of the concrete classes, I've always abstracted them behind my own interface and wrapper class.

Why?

To gain the most test coverage of all the user requirements, especially when those requirements cover logging.

When you have all of your logging going via an interface rather than a concrete class, it becomes very easy to provide a mock object with the logging interface (pick your choice of how you do that, dependency injection etc..) to record all of the logging calls made during a particular test scenario.

These can then be asserted and if a code change breaks the logging your unit tests will cover it.

If NLog or log4Net were to provide an interface for their loggers, then I wouldn't need to go to the effort of providing an interface and wrapper class, since I could just mock their interface for the tests.

Share Improve this answer Follow

answered Aug 28, 2013 at 9:11

nrjohnstone
818 ● 11 ● 17

3

In my project I use `System.Diagnostics.Trace.TraceError(...)`, `System.Diagnostics.Debug.Print(...)` as facade for logging. For organizing (writing) logs I use NLog, i.e. in app.config I have configuration for NLog and redirection of .net's trace to NLog.

```
<nlog xmlns="http://www.nlog-
project.org/schemas/NLog.xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance">
   <targets>
     <target name="file" xsi:type="File"
      layout="${longdate:universalTime=true}Z
[${threadid}]
${pad:padding=5:inner=${level:uppercase=true}}
${logger} ${message}"

fileName="${basedir}/App_Data/logfile.txt"...
   </targets>
</nlog>
<system.diagnostics>
  <trace>
```

```
    <listeners>
      <add name="nlog"
type="NLog.NLogTraceListener, NLog" />
    </listeners>
  </trace>
</system.diagnostics>
```

This does not bundle me to any logger. When I ship my components to customers they can use any logger they like. Using particular logger in application could cause problems i.e. you can use nlog but your customers use log4net.

Share Improve this answer

Follow

answered Sep 10, 2014 at 9:34

Stanislav Berkov

6,287 ● 2 ● 33 ● 39

1

It's not the magic number, it depends on how flexible you wanna be. If you think about changing the log facade one day you should write a facade facade. If you think about changing only the log you need one facade. If you don't think about nothing don't use facade.

The disadvantage as you said is the special abilities. If you are using them write your own facade only.

Share Improve this answer

Follow

answered Sep 29, 2010 at 19:54

Chen Kinnrot

21k ● 17 ● 81 ● 142

0

In this example you only need one level of abstraction to be able to swap out your logger.

What additional advantage would being able to swap out your logging facade get you?

Share Improve this answer

Follow

answered Sep 29, 2010 at 19:51

Christopher Edwards

6,649 ● 8 ● 47 ● 57

If a better logging facade came along or if the logging facade's project went stale, I would be able to swap it out.
– Brian Sep 29, 2010 at 19:52

:) True, but IMHO YAGNI, and anyway sometimes you have to accept that at some stage in the future you're just gonna have to fire up the code editor and make some changes. That said if think you have need then do it. But I never want to see a third party logging facade facade facade project...
– Christopher Edwards Sep 29, 2010 at 20:05

0

It is usable in protocol for plugin system. Instead of saying `use Some3rdParty.dll and MyPlugingApi.dll` I would document only `MyPlugingApi.dll`. Log facade interface will suggest and document some usage that probably will lead to readable logs and good enough logging performance. And of course doing changes will not lead to changes to plugin API.

Why need to change implementation? This can happen if current appears to slow to start out of config or slow to

write entries, desire to expand to cut down version of .NET which does not have needed 3rd party, integrating with other codebase which uses other log writers.

I also have written yet another facade which is child of thoughts in this answer.

Share Improve this answer

Follow

answered Apr 23, 2014 at 10:05

Dzmitry Lahoda

940 ● 1 ● 14 ● 35

0

I'm by no means an expert myself, but in our group, the value of the facade is not in giving us the ability to change the logging framework. True, that is something that we get, but we're in the category who is very unlikely to change our framework.

In our case, we are using a facade to tailor the logging **interface** to the needs of our application. We found that in all the semantic frameworks we looked at, they were still too focused on what we call the "forensics" model of logging--someone digging through the logs looking for some line of output in an attempt to analyze some event.

While this is a use case for us as well, it is actually not our primary use case. We want more of an *instrumentation* framework than a logging one, which will allow us to report on things of interest--even ones we haven't thought of at implementation time.

For example, our application doesn't need a "message" to accompany an event; instead, our "logger" will accept enums defining the event type and state objects representing specifics (eg. timestamps or other values) and will serialize these to facilitate reporting, perf analysis, business value metrics and so on, all in addition to supporting traditional forensics. (We recognize that we are, in fact, making the traditional forensics a little harder for the benefit of a simple-to-use-and-understand logging interface that increases the likelihood that we will actually use it and use it more often).

So to give you a summary answer succinctly, here's a roughly ranked order of the benefits we feel we're getting from using a logging facade.

1. **Consistent, purpose-built interface** facilitating instrumentation (as opposed to just traditional logging, as discussed above)
2. **Mockable test points**
3. **Injectable logger implementations** suitable for everything from local development to AWS S3 or DB connections, depending on deployment (our app uses Autofac with constructor dependency injection)
4. **Substitutable logger frameworks** enabling us to change to a different logger framework in the future if we want to. (Incidentally, we don't forsee this happening, so, on its own, wouldn't have convinced us to use a facade.)

And finally, 0 facades would lose these benefits, and 2 facades would not add to any of these benefits, so that's why 1 facade is the right number for us.

Great question, @brian! :)

Share Improve this answer Follow

edited Jan 31, 2017 at 20:57

answered Jan 31, 2017 at 20:51

U007D

6,288 ● 2 ● 40 ● 47