Conditional logging with minimal cyclomatic complexity

Asked 16 years, 3 months ago Modified 4 years, 7 months ago

Viewed 30k times

78

After reading "What's your/a good limit for cyclomatic complexity?", I realize many of my colleagues were quite annoyed with this new QA policy on our project: no more 10 cyclomatic complexity per function.

Meaning: no more than 10 'if', 'else', 'try', 'catch' and other code workflow branching statement. Right. As I explained in 'Do you test private method?', such a policy has many good side-effects.

But: At the beginning of our (200 people - 7 years long) project, we were happily logging (and no, we can not easily delegate that to some kind of 'Aspect-oriented programming' approach for logs).

```
myLogger.info("A String");
myLogger.fine("A more complicated String");
...
```

And when the first versions of our System went live, we experienced huge memory problem not because of the logging (which was at one point turned off), but because of the *log parameters* (the strings), which are always calculated, then passed to the 'info()' or 'fine()' functions, only to discover that the level of logging was 'OFF', and that no logging were taking place!

So QA came back and urged our programmers to do conditional logging. Always.

```
if(myLogger.isLoggable(Level.INFO) {
myLogger.info("A String");
if(myLogger.isLoggable(Level.FINE) {
myLogger.fine("A more complicated String");
...
```

But now, with that 'can-not-be-moved' 10 cyclomatic complexity level per function limit, they argue that the various logs they put in their function is felt as a burden, because each "if(isLoggable())" is counted as +1 cyclomatic complexity!

So if a function has 8 'if', 'else' and so on, in one tightly-coupled not-easily-shareable algorithm, and 3 critical log actions... they breach the limit even though the conditional logs may not be *really* part of said complexity of that function...

How would you address this situation ?
I have seen a couple of interesting coding evolution (due to that 'conflict') in my project, but I just want to get your thoughts first.

Thank you for all the answers.
I must insist that the problem is not 'formatting' related,

but 'argument evaluation' related (evaluation that can be very costly to do, just before calling a method which will do nothing)
So when a wrote above "A String", I actually meant aFunction(), with aFunction() returning a String, and being a call to a complicated method collecting and computing all kind of log data to be displayed by the logger... or not (hence the issue, and the *obligation* to use conditional logging, hence the actual issue of artificial increase of 'cyclomatic complexity'...)

I now get the 'variadic function' point advanced by some of you (thank you John).
Note: a quick test in java6 shows that my varargs function does evaluate its arguments before being called, so it can not be applied for function call, but for 'Log retriever object' (or 'function wrapper'), on which the toString() will only be called if needed. Got it.

I have now posted my experience on this topic.
I will leave it there until next Tuesday for voting, then I will select one of your answers.
Again, thank you for all the suggestions :)

language-agnostic **logging** **coding-style** **cyclomatic-complexity**

Share
Improve this question
Follow

edited Apr 29, 2020 at 13:30

community wiki
11 revs, 4 users 77%
VonC

6 arbitrary cyclomatic complexity requirements + complicated (i.e., useful) logging statements = incentive to write crappy logging code – mob Oct 20, 2009 at 22:31

A bit late, but anyway. Depending on the language, you might be able to hide the `if` statement inside a method/macro. For example, if this is C/C++, macros would obviously help, although using them might violate some other QA policy. C# generic methods (i.e. a bunch of `Info<T>`, `Info<T1,T2>`, `Info<T1,T2,T3>` would also allow you to avoid this issue since parameters would be passed to the method without boxing/converting to string, and they are easily inlined if they are static (which applies to extension methods too). – vgru Aug 12, 2017 at 12:05

@Groo I agree. At the time (Sept. 19th, 2008, 4 days after the official public release of the Stack Overflow website), I was asking with Java in mind. – VonC Aug 12, 2017 at 12:20

@VonC: yes, crap, I realized this afterward. :) I saw the "edited" timestamp from 2017 so I didn't check the original date. – vgru Aug 12, 2017 at 12:23

12 Answers

Sorted by: Highest score (default)

73

With current logging frameworks, the question is moot

Current logging frameworks like slf4j or log4j 2 don't require guard statements in most cases. They use a parameterized log statement so that an event can be logged unconditionally, but message formatting only occurs if the event is enabled. Message construction is performed as needed by the logger, rather than pre-emptively by the application.

If you have to use an antique logging library, you can read on to get more background and a way to retrofit the old library with parameterized messages.

Are guard statements really adding complexity?

Consider excluding logging guards statements from the cyclomatic complexity calculation.

It could be argued that, due to their predictable form, conditional logging checks really don't contribute to the complexity of the code.

Inflexible metrics can make an otherwise good programmer turn bad. Be careful!

Assuming that your tools for calculating complexity can't be tailored to that degree, the following approach may offer a work-around.

The need for conditional logging

I assume that your guard statements were introduced because you had code like this:

```
private static final Logger log =
Logger.getLogger(MyClass.class);

Connection connect(Widget w, Dongle d, Dongle alt)
  throws ConnectionException
{
  log.debug("Attempting connection of dongle " + d
+ " to widget " + w);
  Connection c;
  try {
    c = w.connect(d);
  } catch(ConnectionException ex) {
    log.warn("Connection failed; attempting
alternate dongle " + d, ex);
    c = w.connect(alt);
  }
  log.debug("Connection succeeded: " + c);
  return c;
}
```

In Java, each of the log statements creates a new `StringBuilder`, and invokes the `toString()` method on each object concatenated to the string. These `toString()` methods, in turn, are likely to create `StringBuilder` instances of their own, and invoke the `toString()` methods of their members, and so on, across a potentially large object graph. (Before Java 5, it was even more expensive, since `StringBuffer` was used, and all of its operations are synchronized.)

This can be relatively costly, especially if the log statement is in some heavily-executed code path. And, written as above, that expensive message formatting occurs even if the logger is bound to discard the result because the log level is too high.

This leads to the introduction of guard statements of the form:

```
  if (log.isDebugEnabled())
    log.debug("Attempting connection of dongle " +
d + " to widget " + w);
```

With this guard, the evaluation of arguments `d` and `w` and the string concatenation is performed only when necessary.

A solution for simple, efficient logging

However, if the logger (or a wrapper that you write around your chosen logging package) takes a formatter and arguments for the formatter, the message construction can be delayed until it is certain that it will be used, while eliminating the guard statements and their cyclomatic complexity.

```
public final class FormatLogger
{

  private final Logger log;

  public FormatLogger(Logger log)
  {
    this.log = log;
  }

  public void debug(String formatter, Object...
args)
  {
    log(Level.DEBUG, formatter, args);
  }
  … &c. for info, warn; also add overloads to log
an exception …

  public void log(Level level, String formatter,
Object... args)
  {
    if (log.isEnabled(level)) {
      /*
       * Only now is the message constructed, and
each "arg"
       * evaluated by having its toString() method
invoked.
       */
      log.log(level, String.format(formatter,
args));
    }
  }

}

class MyClass
{
```

Now, **none of the cascading `toString()` calls with their buffer allocations will occur** unless they are necessary! This effectively eliminates the performance hit that led to the guard statements. One small penalty, in Java, would be auto-boxing of any primitive type arguments you pass to the logger.

The code doing the logging is arguably even cleaner than ever, since untidy string concatenation is gone. It can be even cleaner if the format strings are externalized (using a `ResourceBundle`), which could also assist in maintenance or localization of the software.

Further enhancements

Also note that, in Java, a `MessageFormat` object could be used in place of a "format" `String`, which gives you additional capabilities such as a choice format to handle cardinal numbers more neatly. Another alternative would be to implement your own formatting capability that invokes some interface that you define for "evaluation", rather than the basic `toString()` method.

Share Improve this answer Follow

edited Jul 10, 2015 at 17:58

answered Sep 19, 2008 at 21:46

erickson
269k ● 59 ● 401 ● 497

+1 for FormatLogger. Having done something similar I can vouch for usefulness of this approach. In some testing that we've done, there is virtually no performance hit when appropriate logging level is turned off. – javashlook Mar 25, 2009 at 9:29

2 log4j version 1.2.17 no longer has Logger.isEnabled(Level) use Logger.isEnabledFor(Level) instead. – CodeMonkeyKing Mar 30, 2015 at 23:30

1 wrapper has a side effect, the line number reported by log4j file will not be your class but rather this line all the time – Kalpesh Soni Jul 10, 2015 at 15:18

@KalpeshSoni Using line number logging in log4j is discouraged because of the large performance penalty. Luckily, log4j is open source, so if you insist on using line number logging, you could adjust the line numbers. Or, you could just use a current version of log4j or slf4j; this answer is

7 years old and less relevant now. – erickson Jul 10, 2015 at 17:10

the answer shows up when you google log4j logging enabled etc keywords :) - sorry i am not sure but what are the more relevant solutions today? how does new log4j or slf4j different if you use wrappers – Kalpesh Soni Jul 10, 2015 at 17:40

34

In Python you pass the formatted values as parameters to the logging function. String formatting is only applied if logging is enabled. There's still the overhead of a function call, but that's minuscule compared to formatting.

```
log.info ("a = %s, b = %s", a, b)
```

You can do something like this for any language with variadic arguments (C/C++, C#/Java, etc).

This isn't really intended for when the arguments are difficult to retrieve, but for when formatting them to strings is expensive. For example, if your code already has a list of numbers in it, you might want to log that list for debugging. Executing `mylist.toString()` will take a while to no benefit, as the result will be thrown away. So you pass `mylist` as a parameter to the logging function, and let it handle string formatting. That way, formatting will only be performed if needed.

Since the OP's question specifically mentions Java, here's how the above can be used:

> I must insist that the problem is not 'formatting' related, but 'argument evaluation' related (evaluation that can be very costly to do, just before calling a method which will do nothing)

The trick is to have objects that will not perform expensive computations until absolutely needed. This is easy in languages like Smalltalk or Python that support lambdas and closures, but is still doable in Java with a bit of imagination.

Say you have a function `get_everything()`. It will retrieve every object from your database into a list. You don't want to call this if the result will be discarded, obviously. So instead of using a call to that function directly, you define an inner class called `LazyGetEverything`:

```
public class MainClass {
    private class LazyGetEverything {
        @Override
        public String toString() {
            return getEverything().toString();
        }
    }

    private Object getEverything() {
        /* returns what you want to .toString() in
the inner class */
    }

    public void logEverything() {
        log.info(new LazyGetEverything());
    }
}
```

In this code, the call to `getEverything()` is wrapped so that it won't actually be executed until it's needed. The logging function will execute `toString()` on its parameters only if debugging is enabled. That way, your code will suffer only the overhead of a function call instead of the full `getEverything()` call.

Share Improve this answer

Follow

edited Jul 15, 2011 at 19:48

Dave

5,163 ● 1 ● 23 ● 27

answered Sep 19, 2008 at 22:06

John Millikin

201k ● 41 ● 215 ● 227

1 That works. It's cool Python supports this directly, but you could do it in a wrapper if your logging toolkit doesn't do it. – erickson Sep 19, 2008 at 22:10

1 Woa... thank you for the cleaning job, John. 3000 rep points do help to do that many downvotes ;) I am not sure about your point thought. Do you mean a and b which can be complicated functions returning String are not evaluated during the log function call ? – VonC Sep 19, 2008 at 22:13

1 No, he means that rather than formatting the message in your own code, pass the data you'd use to do it to the logger, which will do the formatting for you, but only if the statement is enabled. This requires a formatter of some sort, rather than string concatenation. – erickson Sep 19, 2008 at 22:29

1 I still not get it, sorry. "pass the data" ??? That mean the data has to be computed/evaluated, in order to be passed to any kind of formatter mechanism, right ? If yes,... I fail to see how

it address my issue (which is not related to format). – VonC Sep 19, 2008 at 22:42

1 Eh? Not sure what you mean by "cleaning". High-rep users have limited amounts of downvotes also, and I haven't edited any posts in this question. ----- VonC: I'll edit my post to show how lazy formatting helps. – John Millikin Sep 19, 2008 at 23:07

6

In languages supporting lambda expressions or code blocks as parameters, one solution for this would be to give just that to the logging method. That one could evaluate the configuration and only if needed actually call/execute the provided lambda/code block. Did not try it yet, though.

Theoretically this is possible. I would not like to use it in production due to performance issues i expect with that heavy use of lamdas/code blocks for logging.

But as always: if in doubt, test it and measure the impact on cpu load and memory.

Share Improve this answer Follow

edited Sep 19, 2008 at 21:56

answered Sep 19, 2008 at 21:50

pointernil
608 ● 8 ● 17

Yeap, good theoretical answer... even though I am still coding in plain old java 1.4.2!!! and do not have access to such neat treat. Still, +1 for the suggestion, since the question is indeed language agnostic. – VonC Sep 19, 2008 at 21:54

4

Thank you for all your answers! You guys rock :)

Now my feedback is not as straight-forward as yours:

Yes, for *one project* (as in 'one program deployed and running on its own on a single production platform'), I suppose you can go all technical on me:

- dedicated 'Log Retriever' objects, which can be pass to a Logger wrapper only calling toString() is necessary
- used in conjunction with a logging variadic function (or a plain Object[] array!)

and there you have it, as explained by @John Millikin and @erickson.

However, this issue forced us to think a little about 'Why exactly we were logging in the first place ?'
Our project is actually 30 different projects (5 to 10 people each) deployed on various production platforms, with asynchronous communication needs and central bus architecture.
The simple logging described in the question was fine for each project *at the beginning* (5 years ago), but since then, we has to step up. Enter the KPI.

Instead of asking to a logger to log anything, we ask to an automatically created object (called KPI) to register an event. It is a simple call (myKPI.I_am_signaling_myself_to_you()), and does not need to be conditional (which solves the 'artificial increase of cyclomatic complexity' issue).

That KPI object knows who calls it and since he runs from the beginning of the application, he is able to retrieve lots of data we were previously computing on the spot when we were logging.
Plus that KPI object can be monitored independently and compute/publish on demand its information on a single and separate publication bus.
That way, each client can ask for the information he actually wants (like, 'has my process begun, and if yes, since when ?'), instead of looking for the correct log file and grepping for a cryptic String...

Indeed, the question 'Why exactly we were logging in the first place ?' made us realize we were not logging just for the programmer and his unit or integration tests, but for a much broader community including some of the final clients themselves. Our 'reporting' mechanism had to be centralized, asynchronous, 24/7.

The specific of that KPI mechanism is way out of the scope of this question. Suffice it to say its proper calibration is by far, hands down, the single most complicated non-functional issue we are facing. It still

does bring the system on its knee from time to time!
Properly calibrated however, it is a life-saver.

Again, thank you for all the suggestions. We will consider them for some parts of our system when simple logging is still in place.
But the other point of this question was to illustrate to you a specific problem in a much larger and more complicated context.
Hope you liked it. I might ask a question on KPI (which, believe or not, is not in any question on SOF so far!) later next week.

I will leave this answer up for voting until next Tuesday, then I will select an answer (not this one obviously ;))

Share Improve this answer Follow

edited Sep 20, 2008 at 7:41

answered Sep 20, 2008 at 7:30

VonC
1.3m ● 558 ● 4.7k ● 5.6k

@Ram 7 and half years laters, KPIs vary from project to project. You have a few listed at en.wikipedia.org/wiki/Performance_indicator#IT_Operations. – VonC Mar 1, 2016 at 7:27

Didn't notice the conversation time! Anyhow thanks for the link1 – RamValli Mar 1, 2016 at 10:08

4

Maybe this is too simple, but what about using the "extract method" refactoring around the guard clause? Your example code of this:

```
public void Example()
{
  if(myLogger.isLoggable(Level.INFO))
      myLogger.info("A String");
  if(myLogger.isLoggable(Level.FINE))
      myLogger.fine("A more complicated String");
  // +1 for each test and log message
}
```

Becomes this:

```
public void Example()
{
   _LogInfo();
   _LogFine();
   // +0 for each test and log message
}

private void _LogInfo()
{
   if(!myLogger.isLoggable(Level.INFO))
      return;

   // Do your complex argument
calculations/evaluations only when needed.
}

private void _LogFine(){ /* Ditto ... */ }
```

Share Improve this answer

Follow

answered Sep 28, 2008 at 1:07

flipdoubt

14.4k ● 17 ● 66 ● 98

Your _LogInfo() and _LogFine() methods don't take any parameters for the log message - and if they did, then the cost of concatenating the parameters would still be incurred. – matt b Oct 7, 2008 at 13:00

Those methods don't take parameters because they calculate the parameters only after testing that the parameters should be calculated. The original poster's point was that the app was being adversely effected by the calculation of unnecessary parameters. I want my rep back! – flipdoubt Oct 7, 2008 at 14:01

See the comment: // Do your complex argument calculations/evaluations only when needed. – flipdoubt Oct 7, 2008 at 14:02

1 You just picked a crappy example. – wowest Feb 24, 2009 at 23:23

Apparently, but I used VonC's example in the original question. – flipdoubt Feb 25, 2009 at 12:35

3

In C or C++ I'd use the preprocessor instead of the if statements for the conditional logging.

Share Improve this answer Follow

answered Sep 19, 2008 at 21:42

Tom Ritter
101k ● 31 ● 142 ● 174

That doesn't make it easy to turn debugging (logging) on in the production code. – Jonathan Leffler Sep 16, 2009 at 5:57

@JonathanLeffler Indeed, because all logging functionality is actually removed before compilation. The release executable does not have any log functionality. – MechMK1 Feb 11, 2015 at 9:57

Tom has a point actually. A macro allows you to "hide" the branching statement (so you are reducing cyclomatic complexity), while the macro parameters are simply replaced without evaluation. It doesn't mean the `if` needs to be hardcoded. – vgru Aug 12, 2017 at 12:07

3

Pass the log level to the logger and let it decide whether or not to write the log statement:

```
//if(myLogger.isLoggable(Level.INFO)
{myLogger.info("A String");
myLogger.info(Level.INFO,"A String");
```

UPDATE: Ah, I see that you want to conditionally create the log string without a conditional statement. Presumably at runtime rather than compile time.

I'll just say that the way we've solved this is to put the formatting code in the logger class so that the formatting only takes place if the level passes. Very similar to a built-in sprintf. For example:

```
myLogger.info(Level.INFO,"A String
%d",some_number);
```

That should meet your criteria.

Share Improve this answer Follow

edited Sep 19, 2008 at 22:06

answered Sep 19, 2008 at 21:45

Greg Miskin

That's what he's doing... before he goes to the work of formatting the message, which can be an expensive waste if the message won't be logged. – erickson Sep 19, 2008 at 21:47

That doesn't solve the problem of the strings being evaluated and using memory despite logging being turned off. – Tom Ritter Sep 19, 2008 at 21:47

Ok. Same answer than @John Millikin. Can you confirm to me than 'some_number' (which actually can be a complicated function returning a string) is **not** evaluated at runtime ? – VonC Sep 19, 2008 at 22:30

2

Conditional logging is evil. It adds unnecessary clutter to your code.

You should always send in the objects you have to the logger:

```
Logger logger = ...
logger.log(Level.DEBUG,"The foo is {0} and the bar
is {1}",new Object[]{foo, bar});
```

and then have a java.util.logging.Formatter that uses MessageFormat to flatten foo and bar into the string to be output. It will only be called if the logger and handler will log at that level.

For added pleasure you could have some kind of expression language to be able to get fine control over how to format the logged objects (toString may not always be useful).

Share Improve this answer Follow

answered Jul 2, 2011 at 22:15

community wiki
simon

2

(source: scala-lang.org)

Scala has a annontation @elidable() that allows you to remove methods with a compiler flag.

With the scala REPL:

```
C:>scala

Welcome to Scala version 2.8.0.final (Java HotSpot(TM) 64-Bit Server VM, Java 1. 6.0_16).
Type in expressions to have them evaluated.
Type :help for more information.
```

```
scala> import scala.annotation.elidable import
scala.annotation.elidable

scala> import scala.annotation.elidable._ import
scala.annotation.elidable._

scala> @elidable(FINE) def logDebug(arg
:String) = println(arg)

logDebug: (arg: String)Unit

scala> logDebug("testing")

scala>
```

With elide-beloset

```
C:>scala -Xelide-below 0

Welcome to Scala version 2.8.0.final (Java
HotSpot(TM) 64-Bit Server VM, Java 1. 6.0_16).
Type in expressions to have them evaluated.
Type :help for more information.

scala> import scala.annotation.elidable import
scala.annotation.elidable

scala> import scala.annotation.elidable._ import
scala.annotation.elidable._

scala> @elidable(FINE) def logDebug(arg
:String) = println(arg)
```

> logDebug: (arg: String)Unit
>
> scala> logDebug("testing")
>
> testing
>
> scala>

See also Scala assert definition

Share Improve this answer Follow

edited Aug 3, 2019 at 5:03

community wiki
4 revs, 3 users 95%
oluies

1

As much as I hate macros in C/C++, at work we have #defines for the if part, which if false ignores (does not evaluate) the following expressions, but if true returns a stream into which stuff can be piped using the '<<' operator. Like this:

```
LOGGER(LEVEL_INFO) << "A String";
```

I assume this would eliminate the extra 'complexity' that your tool sees, and also eliminates any calculating of the string, or any expressions to be logged if the level was not reached.

Share Improve this answer Follow

edited Sep 27, 2008 at 18:27

answered Sep 19, 2008 at 21:47

quamrana
39.3k ● 13 ● 55 ● 76

Hey, if this avoid the evaluation of "A String" at runtime, it is a valid answer (I am just a little rusty on my C/C++ level) – VonC Sep 19, 2008 at 21:56

What would happen if logging is disabled? The resulting object would still need `operator<<`, and so the output operator chain would still be executed. I do not think this answer would help any. – John Millikin Sep 19, 2008 at 22:08

1 The question includes something like: if(myLogger.isLoggable(Level.INFO) myLogger.info("A String"); Now if we just add #define LOGGER(LEVEL) if (myLogger.isLoggable(LEVEL)) myLogger() then assuming 'myLogger()' returns a stream you win all ways. (except you're using macros) – quamrana Sep 20, 2008 at 22:01

1

Here is an elegant solution using ternary expression

logger.info(logger.isInfoEnabled() ? "Log Statement goes here..." : null);

Share Improve this answer

Follow

answered Sep 16, 2009 at 5:38

community wiki
Muhammad Atif Riaz

5 Using the ternary operator still increases cyclomatic complexity. – mob Oct 20, 2009 at 15:20

1

Consider a logging util function ...

```
void debugUtil(String s, Object… args) {
   if (LOG.isDebugEnabled())
       LOG.debug(s, args);
   }
);
```

Then make the call with a "closure" round the expensive evaluation that you want to avoid.

```
debugUtil(“We got a %s”, new Object() {
       @Override String toString() {
       // only evaluated if the debug statement is
executed
           return
expensiveCallToGetSomeValue().toString;
       }
    }
);
```

Share Improve this answer

Follow

edited Nov 22, 2011 at 14:05

community wiki
2 revs, 2 users 91%
johnlon