Spartan Programming

Asked 16 years, 3 months ago Modified 14 years ago Viewed 3k times

10

I really enjoyed Jeff's post on Spartan Programming. I agree that code like that is a joy to read. Unfortunately, I'm not so sure it would necessarily be a joy to work with.

For years I have read about and adhered to the "one-expression-per-line" practice. I have fought the good fight and held my ground when many programming books countered this advice with example code like:

```
while (bytes = read(...))
{
   ...
}

while (GetMessage(...))
{
   ...
}
```

Recently, I've advocated one expression per line for more practical reasons - debugging and production support. Getting a log file from production that claims a NullPointer exception at "line 65" which reads:

```
ObjectA a =
getTheUser(session.getState().getAccount().getAccoun
```

is frustrating and entirely avoidable. Short of grabbing an expert with the code that can choose the "most likely" object that was null ... this is a real practical pain.

One expression per line also helps out quite a bit while stepping through code. I practice this with the assumption that most modern compilers can optimize away all the superfluous temp objects I've just created ...

I try to be neat - but cluttering my code with explicit objects sure feels laborious at times. It does not generally make the code easier to browse - but it really has come in handy when tracing things down in production or stepping through my or someone else's code.

What style do *you* advocate and can you rationalize it in a practical sense?

language-agnostic **styles** **spartan**

Share

Improve this question

Follow

edited Sep 9, 2008 at 4:33

travis

36.4k ● 21 ● 72 ● 97

asked Sep 9, 2008 at 2:05

Luther Baker

7,341 ● 13 ● 49 ● 64

8 Answers

Sorted by: Highest score (default)

In **The Pragmatic Programmer** Hunt and Thomas talk about a study they term the Law of Demeter and it focuses on the coupling of functions to modules other than there own. By allowing a function to never reach a 3rd level in it's coupling you significantly reduce the number of errors and increase the maintainability of the code.

So:

```
ObjectA a =
getTheUser(session.getState().getAccount().get
AccountNumber());
```

Is close to a felony because we are 4 objects down the rat hole. That means to change something in one of those objects I have to know that you called this whole stack right here in this very method. What a pain.

Better:

```
Account.getUser();
```

Note this runs counter to the expressive forms of programming that are now really popular with mocking software. The trade off there is that you have a tightly coupled interface anyway, and the expressive syntax just makes it easier to use.

Share Improve this answer

Follow

edited Feb 9, 2010 at 2:34

Jim Ferrans

31k • 12 • 58 • 83

answered Sep 9, 2008 at 3:26

Dan Blair

2,381 • 3 • 21 • 32

The fluent APIs you're talking about generally don't walk down a tree of stateful objects (like your first example does), rather, they return the same object or new internal objects encapsulating state. So while they violate the letter of the LoD, they don't violate its spirit. – kyoryu Feb 9, 2010 at 3:09

@Kyoryu, I agree with your statement, but the question was focused on 1 statement per line. The fluent API's still make it a little harder to debug because you are executing multiple operations on a single line. – Dan Blair Feb 9, 2010 at 15:35

1 I get your point entirely. Good fluent interfaces use the 'multiple statements' as simply a better way to achieve a single logical statement, rather than actually being multiple, atomic calls. For your example, I'd even argue that Account.GetUser() isn't a particularly good example, as one would typically then act on the user. I'd rather see Account.ActualOperation(). (still upvoted as I think we agree in general, we're quibbling on the 2%) – kyoryu Feb 14, 2010 at 20:19

5

I think the ideal solution is to find a balance between the extremes. There is no way to write a rule that will fit in all situations; it comes with experience. Declaring each intermediate variable on its own line will make reading the code more difficult, which will also contribute to the

difficulty in maintenance. By the same token, debugging is much more difficult if you inline the intermediate values.

The 'sweet spot' is somewhere in the middle.

Share Improve this answer

Follow

answered Sep 9, 2008 at 2:46

pkaeding

37.6k ● 31 ● 106 ● 142

4

One expression per line.

There is no reason to obfuscate your code. The extra time you take typing the few extra terms, you save in debug time.

Share Improve this answer

Follow

answered Sep 9, 2008 at 2:08

jjnguy

139k ● 53 ● 297 ● 326

3

I tend to err on the side of readability, not necessarily debuggability. The examples you gave should definitely be avoided, but I feel that judicious use of multiple expressions can make the code more concise and comprehensible.

Share Improve this answer

Follow

answered Sep 9, 2008 at 3:01

amrox

6,237 ● 4 ● 39 ● 57

2

I'm usually in the "shorter is better" camp. Your example is good:

```
ObjectA a =
getTheUser(session.getState().getAccount().getAccoun
```

I would cringe if I saw that over four lines instead of one--I don't think it'd make it easier to read or understand. The way you presented it here, it's clear that you're digging for a single object. This isn't better:

```
obja State = session.getState();
objb Account = State.getAccount();
objc AccountNumber = Account.getAccountNumber();
ObjectA a = getTheUser(AccountNumber);
```

This is a compromise:

```
objb Account = session.getState().getAccount();
ObjectA a =
getTheUser(Account.getAccountNumber());
```

but I still prefer the single line expression. Here's an anecdotal reason: it's difficult for me to reread and error-check the 4-liner right now for dumb typos; the single line doesn't have this problem because there are simply fewer characters.

Share Improve this answer

Follow

answered Sep 9, 2008 at 3:44

Michael Haren

108k ● 41 ● 171 ● 206

1 But if a NullPointerException is raised with ObjectA a = getTheUser(session.getState().getAccount().getAccountNumber()); which object in the train is null? Session, State, Account, AccountNumber or the User being returned from the original funciton? – Ian Sep 9, 2008 at 10:09

2

```
ObjectA a =
getTheUser(session.getState().getAccount().getAccoun
```

This is a bad example, probably because you just wrote something from the top of your head. You are assigning, to variable named `a` of type `ObjectA`, the return value of a function named `getTheUser`.
So let's assume you wrote this instead:

```
User u =
getTheUser(session.getState().getAccount().getAccoun
```

I would break this expression like so:

```
Account acc = session.getState().getAccount();
User user = getTheUser( acc.getAccountNumber() );
```

My reasoning is: how would I think about what I am doing with this code?
I would probably think: "first I need to get the account from the session and then I get the user using that account's number".

The code should read the way you think. Variables should refer to the main entities involved; not so much to their properties (so I wouldn't store the account number in a variable).

A second factor to have in mind is: will I ever need to refer to this entity again in this context?
If, say, I'm pulling more stuff out of the session state, I would introduce `SessionState state = session.getState()`.

This all seems obvious, but I'm afraid I have some difficulty putting in words why it makes sense, not being a native English speaker and all.

Share Improve this answer Follow

answered Nov 28, 2010 at 13:54

Zecc
4,330 ● 22 ● 17

0

Maintainability, and with it, readability, is king. Luckily, shorter very often means more readable.

Here are a few tips I enjoy using to slice and dice code:

- **Variable names**: how would you describe this variable to someone else on your team? You would *not* say "the numberOfLinesSoFar integer". You would say "numLines" or something similar - comprehensible and short. Don't pretend like the maintainer doesn't know the code at all, but make sure you yourself could figure out what the variable

is, even if you forgot your own act of writing it. Yes, this is kind of obvious, but it's worth more effort than I see many coders put into it, so I list it first.

- **Control flow**: Avoid lots of closing clauses at once (a series of }'s in C++). Usually when you see this, there's a way to avoid it. A common case is something like

:

```
if (things_are_ok) {
  // Do a lot of stuff.
  return true;
} else {
  ExpressDismay(error_str);
  return false;
}
```

can be replaced by

```
if (!things_are_ok) return
ExpressDismay(error_str);
// Do a lot of stuff.
return true;
```

if we can get ExpressDismay (or a wrapper thereof) to return false.

Another case is:

- **Loop iterations**: the more standard, the better. For shorter loops, it's good to use one-character iterators

when the variable is never used except as an index into a single object.

The particular case I would argue here is against the "right" way to use an STL container:

```
for (vector<string>::iterator a_str =
my_vec.begin(); a_str != my_vec.end(); ++a_str)
```

is a lot wordier, and requires overloaded pointer operators *a_str or a_str->size() in the loop. For containers that have fast random access, the following is a lot easier to read:

```
for (int i = 0; i < my_vec.size(); ++i)
```

with references to my_vec[i] in the loop body, which won't confuse anyone.

Finally, I often see coders take pride in their line number counts. But it's not the line numbers that count! I'm not sure of the best way to implement this, but if you have any influence over your coding culture, I'd try to shift the reward toward those with compact classes :)

Share Improve this answer

Follow

answered Sep 9, 2008 at 9:48

Tyler

28.9k ● 12 ● 93 ● 108

Good explanation. I think this is version of the general Divide and Conquer mentality.

0

Share Improve this answer

Follow

answered Sep 9, 2008 at 10:25

epatel

46k ● 17 ● 111 ● 144