**Abbreviations**

HID - High Information Density

LID - Low Information Density

HIV - High Information Volume

LIV - Low Information Volume

HID/HIV - Those who can handle both HID and HIV well

HID/LIV - Those who can handle HID well but can only handle LIV well

LID/HIV - Those who can only handle LID well but can handle HIV well

LID/LIV - Those who can only handle LID and LIV well

**TL;DR(The Whole Article Takes About 30 Minutes To Read In Full Depth)**

**Information Density**

**A small piece of information representation referring to a large piece of information content has HID, whereas a large piece of information representation referring to a small piece of information content has LID. Unfortunately, different programmers have different capacities on facing information density.**

In general, those who can handle very HID well will prefer very terse codes, as it'll be more effective and efficient to both write and read them that way for such software engineers, while writing and reading verbose codes are just wasting their time in their perspectives; Those who can only handle very LID well will prefer very verbose codes, as it'll be easier and simpler to both write and read them that way for such software engineers, while writing and reading terse codes are just too complicated and convoluted in their perspectives. **Ideally, we should be able to handle very HID well while still being very tolerant towards LID, so we'd be able to work well with codes having all kinds of information density.** Unfortunately, very effective and efficient software engineers are generally very intolerant towards extreme ineffectiveness or inefficiencies, so all we can do is to try hard.

**Information Volume**

**A code chunk having a large piece of information content that aren't abstracted away from that code chunk has HIV, whereas a code chunk having only a small piece of information content that aren't abstracted away from that code chunk has LIV. Unfortunately, different software engineers have different capacities on facing information volume, so it seems that the best way's to find a happy medium that can break a very long function into fathomable chunks on one hand, while still keeping the function call stack manageable on the other.**

In general, those who can handle very HIV well will prefer very long functions, as it'll be more effective and efficient to draw the full picture without missing any nontrivial relevant detail that way for such software engineers, while writing and reading very short functions are just going the opposite directions in their perspectives; Those who can only handle very LIV well will prefer very short functions, as it'll be easier and simpler to reason about well-defined abstractions(as long as they don't [leak](https://en.wikipedia.org/wiki/Leaky_abstraction) in nontrivial ways) that way for such software engineers, while writing and reading long functions are just going the opposite directions in their perspectives. **Ideally, we should be able to handle very HIV well while still being very tolerant towards LIV, so we'd be able to work well with codes having all kinds of information volume.** Unfortunately, very effective and efficient software engineers are generally very intolerant towards extreme ineffectiveness or inefficiencies(especially when those small function abstractions do [leak](https://en.wikipedia.org/wiki/Leaky_abstraction) in nontrivial ways), so all we can do is to try hard.

**Combining Information Density With Information Volume**

**While information density and volume are closely related, there's no strict implications from one to the other, meaning that there are different combinations of these 2 factors and the resultant style can be very different from each other.** For instance, HID doesn't imply LIV nor vice versa, as it's possible to write a very terse long function and a very verbose short function; LID doesn't imply HIV nor vice versa for the very same reasons. In general, the following largely applies to most codebases, even when there are exceptions:

**Very HID + HIV =** [**Massive Ball**](https://en.wikipedia.org/wiki/Big_ball_of_mud) **Of Complicated And Convoluted** [**Spaghetti**](https://en.wikipedia.org/wiki/Spaghetti_code)[**Legacy**](https://en.wikipedia.org/wiki/Legacy_code)

**Very HID + LIV = Otherwise High Quality Codes That Are Hard To Fathom At First**

**Very LID + HIV = Excessively Verbose Codes With Tons Of Redundant Boilerplate**

**Very LID + LIV = Too Many Small Functions With The Call Stacks Being Too Deep**

**Teams With Programmers Having Different Styles**

**It seems to me that many coding standard/style conflicts can be somehow explained by the conflicts between HID and LID, and those between HIV and LIV, especially when both sides are being more and more extreme.** The combinations of these conflicts may be:

**Very HID/HIV + HID/LIV = Too Little Architecture vs Too Weak To Fathom Codes**

**Very HID/HIV + LID/HIV = Being Way Too Complex vs Doing Too Little Things**

**Very HID/HIV + LID/LIV =** [**Over-Optimization**](https://en.wikipedia.org/wiki/Program_optimization#When_to_optimize) **Freak vs** [**Over-Engineering**](https://en.wikipedia.org/wiki/Overengineering) **Freak**

**Very HID/LIV + LID/HIV = Too Concise/Organized vs Too Messy/Verbose**

**Very HID/LIV + LID/LIV = Too Hard To Read At First vs Too Ineffective/Inefficient**

**Very LID/HIV + LID/LIV = Too Beginner Friendly vs Too Flexible For Impossibles**

**﻿Conclusions**

Of course, one doesn't have to go for the HID, LID, HIV or LIV extremes, as there's quite some middle grounds to play with. In fact, **I think the best of the best software engineers should deal with all these extremes well while still being able to play with the middle grounds well, provided that such an exceptional software engineer can even exist at all.** Nevertheless, it's rather common to work with at least some of the software engineers falling into at least 1 extremes, so we should still know how to work well with them. After all, nowadays most of the real life business codebase are about teamwork but not lone wolves.

By exploring the importance of information density, information volume and their relationships, I hope that this article can help us think of some aspects behind codebase readability and the nature of conflicts about it, and that we can be more able to deal with more different kinds of codebase and software engineers better. I think that it's more feasible for us to be able to read codebase with different information density and volume than asking others and the codebase to accommodate with our information density/volume limitations.

Also, this article actually implies that readability's probably a complicated and convoluted concept, as it's partially objective at large(e.g.: the existence of consistent formatting and meaningful naming) and partially subjective at large(e.g.: the ability to handle different kinds of information density and volume for different software engineers). **Maybe many avoidable conflicts involving readability stems from the tendency that many software engineers treat readability as easy, simple and small concept that are entirely objective.**

**Information Density**

**A Math Analogy**

Consider the following math formula that are likely learnt in high school(Euler's Formula):

Most of those who've studied high school math well should immediately fathom this, but for those who don't, you may want to try to fathom this text equivalent, which is more verbose:

The Euler number to the power of (the imaginary unit multiplied by theta in radian) equals cosine theta in radian plus the imaginary unit multiplied by sine theta in radian

I hope that those who can't fathom the above formula can at least fathom the above text :)

This brings the importance of information density: **A small piece of information representation referring to a large piece of information content has HID, whereas a large piece of information representation referring to a small piece of information content has LID**. For instance, the above formula has HID whereas the above text has LID.

In this example, those who're good at math in general and high school math in particular will likely prefer the formula over the text equivalent as they can probably fathom the former instantly while feeling that the latter's just wasting their time; Those who're bad at math in general and high school math in particular will likely prefer the text equivalent over the formula as they might not even know the fact that cisx is the short form of cosx + isinx.

For those who can handle HID well, even if they don't know what Euler number is at all, they should still be able to deduce these corollaries within minutes if they know what cisx is:

But for those who can only handle LID well, they'll unlikely be able to know what's going on at all, even if they know how to use the binomial theorem and the truncation operator.

Now let's try to fathom this math formula that can be fathomed using just high school math:

While it doesn't involve as much math knowledge nor concepts as those in the Euler's Formula, I'd guess that only those who're really, really exceptional in high school math and math in general can fathom this within seconds, let alone instantly, all because of this formula having such a ridiculously HID. If you can really fathom this instantly, then I'd think that you can really handle very HID very well, especially when it comes to math :D

So what if we try to explain this by text? I'd come up with the following try:

(The summation of m variables from x1 to xm) to the power of n equals the summation of (n elements, each being the combination of selecting r elements from n - 1 elements, where r is the outermost summation counter from 0 to n - 1, multiplied by the summation of (m elements, each being xi to the power of n - r, where i is the middle summation counter from 1 to m, multiplied by (the summation of m variables from x1 to xm except xi) to the power of r))

Maybe you can finally fathom what this formula is, but still probably not what it really means nor how to use it meaningfully, let alone deducing any useful corollary. However, with the text version, at least we can clearly see just how high the information density is in that formula, as even the information density for the text version isn't actually anything low.

These 2 math examples aim to show that, HID, as long as being kept in moderation, is generally preferred over the LID counterparts. But once the information density becomes too unnecessarily and unreasonably high, the much more verbose versions seeming to be too verbose is actually preferred in general, especially when their information density isn't low.

**Some Examples Showing HID vs LID**

There are programming parallels to the above math analogy: terse and verbose codes. **Unfortunately, different programmers have different capacities on facing information density, just like different people have different capacities on fathoming math.**

For instance, the ternary operator is a very obvious terse example on this(Javascript ES5):

var x = condition1 ? value1 : condition2 ? value2 : value3;

Whereas a verbose if/else if/else equivalent can be something like this:

var x;

if (condition1 === true) {

x = value1;

} else if (condition2 === true) {

x = value2;

} else {

x = value3;

}

Those who're used to read and write terse codes will likely like the ternary operator version as the if/else if/else version will likely be just too verbose for them; Those who're used to read and write verbose codes will likely like the if/else if/else version as the ternary operator version will likely be just too terse for them(I've seen production codes with if (variable === true), so don't think that the if/else if/else version can only be totally made up examples). In this case, I've worked with both styles, and I guess that most programmers can handle both.

Similarly, Javascript and some other languages support short circuit evaluation, which is also a terse style. For instance, the || and && operators can be short circuited this way:

return isValid && (array || []).concat(object || canUseDefault && default);

Where a verbose equivalent can be something like this(it's probably too verbose anyway):

var returnedValue;

if (isValid === true) {

var returnedArray;

var isValidArray = (array !== null) && (array !== undefined);

if (isValidArray === true) {

returnedArray = array;

} else {

returnedArray = [];

}

var pushedObject;

var isValidObject = (object !== null) && (object !== undefined);

if (isValidObject === true) {

pushedObject = object;

} else if (canUseDefault === true) {

pushedObject = default;

} else {

pushedObject = canUseDefault;

}

if (Array.isArray(pushedObject) === true) {

returnedArray = returnedArray.concat(pushedObject);

} else {

returnedArray = returnedArray.concat([pushedObject]);

}

returnedValue = returnedArray;

} else {

returnedValue = isValid;

}

return returnedValue;

Clearly the terse version has very HID while the verbose version has very LID. Those who can handle HID well will likely fathom the terse version instantly while needing minutes just to fathom what the verbose version's really trying to achieve and why it's not written in the terse version to avoid wasting time to read so much code doing so little meaningful things; Those who can only handle LID well will likely fathom the verbose version within minutes while probably giving up after trying to fathom the terse version for seconds and wonder what's the point of being concise when it's doing just so many things in just 1 line. In this case, I seriously suspect whether anyone fathoming Javascript will ever write in the verbose version at all, when the terse version is actually one of the popular idiomatic styles.

Now let's try to fathom this really, really terse codes(I hope you won't face this in real life):

for (var texts = [], num = min; num <= max; num += increment) {

    var primeMods = primes.map(function(prime) { return num % prime; });

texts.push(primeMods.reduce(function(text, mod, i) {

return (text + (mod || words[i])).replace(mod, "");

}, "") || num);

}

return texts.join(textSeparator);

If you can fathom this within seconds or even instantly, then I'd admit that you can really handle ridiculously HID exceptionally well. However, adding these lines will make it clear:

var min = 1, max = 100, increment = 1;

var primes = [3, 5], words = ["Fizz", "Buzz"], textSeparator = "\n";

So all it's trying to do is the very, very popular [Fizz Buzz](https://en.wikipedia.org/wiki/Fizz_buzz) programming test in a ridiculously terse way. So let's try this much more verbose version of this [Fizz Buzz](https://en.wikipedia.org/wiki/Fizz_buzz) programming test:

var texts = [];

for (var num = min; num <= max; num = num + increment) {

    var text = "";

var primeCount = primes.length;

for (var i = 0; i < primeCount; i = i + 1) {

var prime = primes[i];

var mod = num % prime;

if (mod === 0) {

var word = words[i];

text = text + word;

}

}

if (text === "") {

texts.push(num);

} else {

texts.push(text);

}

}

return texts.join(textSeparator);

Even those who can handle very HID well should still be able to fathom this verbose version within seconds, so do those who can only handle very LID well. Also, considering the inherent complexity of this generalized [Fizz Buzz](https://en.wikipedia.org/wiki/Fizz_buzz), the verbose version doesn't have much boilerplate, even when compared to the terse version, so I don't think those who can handle very HID well will complain about the verbose version much. On the other hand, I doubt whether those who can only handle very LID well can even fathom the terse version, let alone in a reasonable amount of time(like minutes), if I didn't tell that it's just [Fizz Buzz](https://en.wikipedia.org/wiki/Fizz_buzz). In this case, I really doubt what's the point of writing in the terse version when I don't see any nontrivial issue in the verbose version(while the terse version's likely harder to fathom).

**Back To The Math Analogy**

Imagine that a mathematician and math professor who's used to teach postdoc math now have to teach high school math to elementary math students(I've heard that a very small amount of parents are so ridiculous to want their elementary children to learn high school math even when those children aren't interested in nor good at math). That's almost mission impossible, but all that teacher can do is to first consolidate the elementary math foundation of those students while fostering their interest in math, then gradually progress to middle school math, and finally high school math once those students are good at middle school math. All those students can do is to work extremely hard to catch up such great hurdles.

Unfortunately, it seems to me that it'd take far too much resources, especially time, when those who can handle very HID well try to teach those who can only handle very LID well to handle HID. Even when those who can only handle very LID well can eventually be nurtured to meet the needs imposed by the codebase, it's still unlikely to be worth it, especially for software teams with very tight budgets, no matter how well intentioned it is.

So should those who can only handle very LID well train up themselves to be able to handle HID? I hope so, but I doubt that it's similar to asking a high school student to fathom postdoc math. While it's possible, I still guess that most of us will think that it's so costly and disproportional just to apply actually basic math formulae that are just written in terse styles; Should those who can handle very HID well learn how to deal with LID well as well? I hope so, but I doubt that's similar to asking mathematicians to abandon their mother tongue when it comes to math(using words instead of symbols to express math). While it's possible, I still guess that most of us will think that it's so excessively ineffective and inefficient just to communicate with those who're very poor at math when discussing about advanced math.

So it seems that maybe those who can handle HID well and those who can only handle LID well should avoid working with each other as much as possible. But that'd mean all these:

1. The current software team must identify whether the majority can handle HID well or can only handle LIV well, which isn't easy to do and most often totally ignored
2. The software engineering job requirement must state that whether being able to deal with HID well will be prioritized or even required, which is an uncommon statement
3. All applicants must know whether they can handle HID well, which is overlooked
4. The candidate screening process must be able to tell who can handle HID well
5. Most importantly, the team must be able to hire enough candidates who can handle HID well, and it's obvious that many software teams just won't be able to do that

Therefore, I don't think it's an ideal or even reasonable solution, even though it's possible.

Alternatively, those who can handle very HID well should try their best to only touch the HID part of the codebase, while those who can only handle very LID well should try their best to only touch the LID part of the codebase. But needless to say, that's way easier said than done, especially when the team's large and the codebase can't be really that modular.

**A Considerable Solution**

With an IDE supporting [collapsing comments](https://en.wikipedia.org/wiki/Code_folding#Collapsing_comments), one can try something like this:

/\*

var returnedValue;

if (isValid === true) {

var returnedArray;

var isValidArray = (array !== null) && (array !== undefined);

if (isValidArray === true) {

returnedArray = array;

} else {

returnedArray = [];

}

var pushedObject;

var isValidObject = (object !== null) && (object !== undefined);

if (isValidObject === true) {

pushedObject = object;

} else if (canUseDefault === true) {

pushedObject = default;

} else {

pushedObject = canUseDefault;

}

if (Array.isArray(pushedObject) === true) {

returnedArray = returnedArray.concat(pushedObject);

} else {

returnedArray = returnedArray.concat([pushedObject]);

}

returnedValue = returnedArray;

} else {

returnedValue = isValid;

}

return returnedValue;

\*/

return isValid && (array || []).concat(object || canUseDefault && default);

Of course it's not practical when the majority of the codebase's so terse that those who can only handle very LID well will struggle most of the time, but those who can handle very HID well can try to do the former some favors when there aren't lots of terse codes for them. **The point of this comment's to be a working compromise between the needs of reading codes effectively and efficiently for those who can handle very HID well, and the needs of fathoming code easily and simply for those who can only handle very LID well.**

**Summary**

In general, those who can handle very HID well will prefer very terse codes, as it'll be more effective and efficient to both write and read them that way for such software engineers, while writing and reading verbose codes are just wasting their time in their perspectives; Those who can only handle very LID well will prefer very verbose codes, as it'll be easier and simpler to both write and read them that way for such software engineers, while writing and reading terse codes are just too complicated and convoluted in their perspectives. **Ideally, we should be able to handle very HID well while still being very tolerant towards LID, so we'd be able to work well with codes having all kinds of information density.** Unfortunately, very effective and efficient software engineers are generally very intolerant towards extreme ineffectiveness or inefficiencies, so all we can do is to try hard.

**Information Volume**

**An Eating Analogy**

Let's say we're ridiculously big eaters who can eat 1kg of meat per meal. But can we eat all that 1kg of meat in just 1 chunk? Probably not, as our mouth just won't be big enough, so we'll have to cut it into digestible chunks. However, can we eat it if it becomes a 1kg of very fine-grained meat powder? Maybe, but that's likely daunting or even dangerous(extremely high risk of severe choking) for most of us. So it seems that the best way's to find a happy medium that works for us, like cutting it into chunks that are just small enough for our mouth to digest. There might still be many chunks but at least they'll be manageable enough.

The same can be largely applied to fathoming codes, even though there are still differences.

Let's say you're reading a well-documented function with 100k lines and none of its business logic are duplicated in the entire codebase(so breaking this function won't help code reuse right now). Unless we're so good at fathoming big functions that we can keep all these 100k lines of implementation details in our head as a whole, reading such a function will likely be daunting or even dangerous(extremely high risk of fathom it all wrong) for most of us, assuming that we can indeed fathom it within a feasible amount of time(like within hours).

On the other hand, if we break that 100k line function into extremely small functions so that the [function call stack](https://en.wikipedia.org/wiki/Call_stack) can be as deep as 100 calls, we'll probably be in really big trouble when we've to debug these functions having bugs that don't have apparently obvious causes nor caught by the current test suite(no test suite can catch all bugs after all). After all, traversing such a deep [call stack](https://en.wikipedia.org/wiki/Call_stack) without getting lost and having to start all over again is like eating tons of very fine-grained meat powders without ever choking severely. Even if we can eventually fix all those bugs with the test suite updated, it'll still unlikely to be done within a reasonable amount of time(talking about days or even weeks when the time budget is tight).

This brings the importance of information volume: **A code chunk having a large piece of information content that aren't abstracted away from that code chunk has HIV, whereas a code chunk having only a small piece of information content that aren't abstracted away from that code chunk has LIV.** For instance, the above 100k line function has HIV whereas the above small functions with deep [call stack](https://en.wikipedia.org/wiki/Call_stack) has LIV.

**So it seems that the best way's to find a happy medium that can break that 100k line function into fathomable chunks on one hand, while still keeping the** [**call stack**](https://en.wikipedia.org/wiki/Call_stack) **manageable on the other.** For instance, if possible, breaking that 100k line function into those in which the largest ones are 1k line functions and the ones with the deepest [call stack](https://en.wikipedia.org/wiki/Call_stack) is 10 calls can be a good enough balance. While fathoming a 1k line function is still hard for most of us, it's at least practical; While debugging functions having [call stacks](https://en.wikipedia.org/wiki/Call_stack) with 10 calls is still time-consuming for most of us, it's at least realistic to be done within a tight budget.

**A Small Example Showing HIV vs LIV**

**Unfortunately, different software engineers have different capacities on facing information volume, just like different people have different mouth size.** Consider the following small example(Some of my Javascript ES5 codes with comments removed):

LIV Version(17 methods with the largest being 4 lines and the deepest [call stack](https://en.wikipedia.org/wiki/Call_stack) being 11) -

$.result = function(note, argObj\_) {

    if (!$gameSystem.satbParam("\_isCached")) {

        return this.\_uncachedResult(note, argObj\_, "WithoutCache");

    }

    return this.\_updatedResult(note, argObj\_);

};

$.\_updatedResult = function(note, argObj\_) {

var cache = this.\_cache.result\_(note, argObj\_);

    if (\_SATB.IS\_VALID\_RESULT(cache)) return cache;

    return this.\_updatedResultWithCache(note, argObj\_);

};

$.\_updatedResultWithCache = function(note, argObj\_) {

    var result = this.\_uncachedResult(note, argObj\_, "WithCache");

    this.\_cache.updateResult(note, argObj\_, result);

    return result;

};

$.\_uncachedResult = function(note, argObj\_, funcNameSuffix) {

    if (this.\_rules.isAssociative(note)) {

        return this.\_associativeResult(note, argObj\_, funcNameSuffix);

    }

    return this.\_nonAssociativeResult(note, argObj\_, funcNameSuffix);

};

$.\_associativeResult = function(note, argObj\_, funcNameSuffix) {

    var partResults = this.\_partResults(note, argObj\_, funcNameSuffix);

    var defaultResult = this.\_pairs.default(note, argObj\_);

return this.\_rules.chainedResult(

partResults, note, argObj\_, defaultResult);

};

$.\_partResults = function(note, argObj\_, funcNameSuffix) {

    var priorities = this.\_rules.priorities(note);

    var funcName = "\_partResult" + funcNameSuffix + "\_";

    var resultFunc = this[funcName].bind(this, note, argObj\_);

    return priorities.map(resultFunc).filter(\_SATB.IS\_VALID\_RESULT);

};

$.\_partResultWithoutCache\_ = function(note, argObj\_, part) {

    return this.\_uncachedPartResult\_(note, argObj\_, part, "WithoutCache");

};

$.\_partResultWithCache\_ = function(note, argObj\_, part) {

    var cache = this.\_cache.partResult\_(note, argObj\_, part);

    if (\_SATB.IS\_VALID\_RESULT(cache)) return cache;

    return this.\_updatedPartResultWithCache\_(note, argObj\_, part);

};

$.\_updatedPartResultWithCache\_ = function(note, argObj\_, part) {

    var result =

            this.\_uncachedPartResult\_(note, argObj\_, part, "WithCache");

    this.\_cache.updatePartResult(note, argObj\_, part, result);

    return result;

};

$.\_uncachedPartResult\_ = function(note, argObj\_, part, funcNameSuffix) {

    var list = this["\_pairFuncListPart" + funcNameSuffix](note, part);

    if (list.length <= 0) return undefined;

return this.\_rules.chainedResult(list, note, argObj\_);

};

$.\_nonAssociativeResult = function(note, argObj\_, funcNameSuffix) {

    var list = this["\_pairFuncList" + funcNameSuffix](note);

    var defaultResult = this.\_pairs.default(note, argObj\_);

return this.\_rules.chainedResult(list, note, argObj\_, defaultResult);

};

$.\_pairFuncListWithoutCache = function(note) {

    return this.\_uncachedPairFuncList(note, "WithoutCache");

};

$.\_pairFuncListWithCache = function(note) {

    var cache = this.\_cache.pairFuncList\_(note);

    return cache || this.\_updatedPairFuncListWithCache(note);

};

$.\_updatedPairFuncListWithCache = function(note) {

    var list = this.\_uncachedPairFuncList(note, "WithCache");

    this.\_cache.updatePairFuncList(note, list);

    return list;

};

$.\_uncachedPairFuncList = function(note, funcNameSuffix) {

    var funcName = "\_pairFuncListPart" + funcNameSuffix;

    return this.\_rules.priorities(note).reduce(function(list, part) {

        return list.concat(this[funcName](note, part));

    }.bind(this), []);

};

$.\_pairFuncListPartWithCache = function(note, part) {

    var cache = this.\_cache.pairFuncListPart\_(note, part);

    return cache || this.\_updatedPairFuncListPartWithCache(note, part);

};

$.\_updatedPairFuncListPartWithCache = function(note, part) {

    var list = this.\_pairFuncListPartWithoutCache(note, part);

    this.\_cache.updatePairFuncListPart(note, part, list);

    return list;

};

$.\_pairFuncListPartWithoutCache = function(note, part) {

    var func = this.\_pairs.pairFuncs.bind(this.\_pairs, note);

    return this.\_cache.partListData(part, this.\_battler).map(func);

};

HIV Version(10 methods with the largest being 12 lines and the deepest [call stack](https://en.wikipedia.org/wiki/Call_stack) being 5) -

$.result = function(note, argObj\_) {

    if (!$gameSystem.satbParam("\_isCached")) {

        return this.\_uncachedResult(note, argObj\_, "WithoutCache");

    }

    var cache = this.\_cache.result\_(note, argObj\_);

    if (\_SATB.IS\_VALID\_RESULT(cache)) return cache;

    // $.\_updatedResultWithCache START

    var result = this.\_uncachedResult(note, argObj\_, "WithCache");

    this.\_cache.updateResult(note, argObj\_, result);

    return result;

    // $.\_updatedResultWithCache END

};

$.\_uncachedResult = function(note, argObj\_, funcNameSuffix) {

    if (this.\_rules.isAssociative(note)) {

        // $.\_associativeResult START

            // $.\_partResults START

        var priorities = this.\_rules.priorities(note);

        var funcName = "\_partResult" + funcNameSuffix + "\_";

        var resultFunc = this[funcName].bind(this, note, argObj\_);

        var partResults =

                priorities.map(resultFunc).filter(\_SATB.IS\_VALID\_RESULT);

            // $.\_partResults END

        var defaultResult = this.\_pairs.default(note, argObj\_);

return this.\_rules.chainedResult(

partResults, note, argObj\_, defaultResult);

        // $.\_associativeResult START

    }

    // $.\_nonAssociativeResult START

    var list = this["\_pairFuncList" + funcNameSuffix](note);

    var defaultResult = this.\_pairs.default(note, argObj\_);

return this.\_rules.chainedResult(list, note, argObj\_, defaultResult);

    // $.\_nonAssociativeResult END

};

$.\_partResultWithoutCache\_ = function(note, argObj\_, part) {

    return this.\_uncachedPartResult\_(note, argObj\_, part, "WithoutCache");

};

$.\_partResultWithCache\_ = function(note, argObj\_, part) {

    var cache = this.\_cache.partResult\_(note, argObj\_, part);

    if (\_SATB.IS\_VALID\_RESULT(cache)) return cache;

    // $.\_updatedPartResultWithCache\_ START

    var result =

            this.\_uncachedPartResult\_(note, argObj\_, part, "WithCache");

    this.\_cache.updatePartResult(note, argObj\_, part, result);

    return result;

    // $.\_updatedPartResultWithCache\_ END

};

$.\_uncachedPartResult\_ = function(note, argObj\_, part, funcNameSuffix) {

    var list = this["\_pairFuncListPart" + funcNameSuffix](note, part);

    if (list.length <= 0) return undefined;

return this.\_rules.chainedResult(list, note, argObj\_);

};

$.\_pairFuncListWithoutCache = function(note) {

    return this.\_uncachedPairFuncList(note, "WithoutCache");

};

$.\_pairFuncListWithCache = function(note) {

    var cache = this.\_cache.pairFuncList\_(note);

    if (cache) return cache;

    // $.\_updatedPairFuncListWithCache START

    var list = this.\_uncachedPairFuncList(note, "WithCache");

    this.\_cache.updatePairFuncList(note, list);

    return list;

    // $.\_updatedPairFuncListWithCache END

};

$.\_uncachedPairFuncList = function(note, funcNameSuffix) {

    var funcName = "\_pairFuncListPart" + funcNameSuffix;

    return this.\_rules.priorities(note).reduce(function(list, part) {

        return list.concat(this[funcName](note, part));

    }.bind(this), []);

};

$.\_pairFuncListPartWithCache = function(note, part) {

    var cache = this.\_cache.pairFuncListPart\_(note, part);

    if (cache) return cache;

    // $.\_updatedPairFuncListPartWithCache START

    var list = this.\_pairFuncListPartWithoutCache(note, part);

    this.\_cache.updatePairFuncListPart(note, part, list);

    return list;

    // $.\_updatedPairFuncListPartWithCache END

};

$.\_pairFuncListPartWithoutCache = function(note, part) {

    var func = this.\_pairs.pairFuncs.bind(this.\_pairs, note);

    return this.\_cache.partListData(part, this.\_battler).map(func);

};

In case you can't fathom what this example's about, you can read this simple flow chart(It doesn't mention the fact that the actual codes also handle whether the cache will be used):

Even though the underlying business logic's easy to fathom, different people will likely react to the HIV and LIV Version differently. Those who can handle very HIV well will likely find the LIV version less readable due to having to unnecessarily traverse all these excessively small methods(the smallest ones being 1 liners) and enduring the highest call stack of 11 calls(from $.result to $.\_pairFuncListPartWithoutCache); Those who can only handle very LIV well will likely find the HIV version less readable due to having to unnecessarily fathom all these excessively mixed implementation details as a single unit in one go from the biggest method with 12 lines and enduring the presence of 3 different [levels of abstractions](https://en.wikipedia.org/wiki/Abstraction_layer) combined just in the biggest and most complex method($.\_uncachedResult).

Bear in mind that it's just a small example which is easy to fathom and simple to explain, so the differences between the HIV and LIV styles and the potential conflicts between those who can handle very HIV well and those who can only handle very LIV well will only be even larger and harder to resolve when it comes to massive real life production codebases.

**Back To The Eating Analogy**

Imagine that the size of the mouth of various people can vary so much that the largest digestible chunk of those with the smallest mouth are as small as a very fine-grained powder in the eyes of those with the largest mouth. Let's say that these 2 extremes are going to eat together sharing the same meal set. How should these meals be prepared? An obvious way's to give them different tools to break these meals into digestible chunks of sizes suiting their needs so they'll respectively use the tools that are appropriate for them, meaning that the meal provider won't try to do these jobs themselves at all. It's possible that those with the smallest mouth will happily break those meals into very fine-grained powders, while those with the largest mouth will just eat each individual food as a whole without much trouble.

Unfortunately, it seems to me that there's still no well battle-tested automatic tools that can effectively and efficiently break a large code chunk into well-defined smaller digestible code chunks with configurable size and complexity without nontrivial side effects, so those who can only handle very LIV well will have to do it manually when having to fathom large functions. Also, even when there's such a tool, such automatic work's still effectively refactoring that function, thus probably irritating colleagues who can handle very HIV well.

So should those who can only handle very LIV well train up themselves to be able to deal with HIV? I hope so, but I doubt that's similar to asking those with very small mouths to increase their mouth size. While it's possible, I still guess that most of us will think that it's so costly and disproportional just to eat foods in chunks that are too large for them; Should those who can handle very HIV well learn how to deal with LIV well as well? I hope so, but I doubt that's similar to asking those with very large mouths to force themselves to eat very fine-grained meat powders without ever choking severely(getting lost when traversing a very deep call stack). While it's possible, I still guess that most of us will think that it's so risky and unreasonable just to eat foods as very fine-grained powders unless they really have no other choices at all(meaning that they should actually avoid these as much as possible).

So it seems that maybe those who can handle HIV well and those who can only handle LIV well should avoid working with each other as much as possible. But that'd mean all these:

1. The current software team must identify whether the majority can handle HIV well or can only handle LIV well, which isn't easy to do and most often totally ignored
2. The software engineering job requirement must state that whether being able to deal with HIV well will be prioritized or even required, which is an uncommon statement
3. All applicants must know whether they can handle HIV well, which is overlooked
4. The candidate screening process must be able to tell who can handle HIV well
5. Most importantly, the team must be able to hire enough candidates who can handle HIV well, and it's obvious that many software teams just won't be able to do that

Therefore, I don't think it's an ideal or even reasonable solution, even though it's possible.

Alternatively, those who can handle very HIV well should try their best to only touch the HIV part of the codebase, while those who can only handle very LIV well should try their best to only touch the LIV part of the codebase. But needless to say, that's way easier said than done, especially when the team's large and the codebase can't be really that modular.

**An Imagined Solution**

Let's say there's an IDE which can display the function calls in the inlined form, like from:

$.result = function(note, argObj\_) {

    if (!$gameSystem.satbParam("\_isCached")) {

        return this.\_uncachedResult(note, argObj\_, "WithoutCache");

    }

    return this.\_updatedResult(note, argObj\_);

};

$.\_updatedResult = function(note, argObj\_) {

var cache = this.\_cache.result\_(note, argObj\_);

    if (\_SATB.IS\_VALID\_RESULT(cache)) return cache;

    return this.\_updatedResultWithCache(note, argObj\_);

};

$.\_updatedResultWithCache = function(note, argObj\_) {

    var result = this.\_uncachedResult(note, argObj\_, "WithCache");

    this.\_cache.updateResult(note, argObj\_, result);

    return result;

};

$.\_uncachedResult = function(note, argObj\_, funcNameSuffix) {

    if (this.\_rules.isAssociative(note)) {

        return this.\_associativeResult(note, argObj\_, funcNameSuffix);

    }

    return this.\_nonAssociativeResult(note, argObj\_, funcNameSuffix);

};

$.\_associativeResult = function(note, argObj\_, funcNameSuffix) {

    var partResults = this.\_partResults(note, argObj\_, funcNameSuffix);

    var defaultResult = this.\_pairs.default(note, argObj\_);

return this.\_rules.chainedResult(

partResults, note, argObj\_, defaultResult);

};

$.\_partResults = function(note, argObj\_, funcNameSuffix) {

    var priorities = this.\_rules.priorities(note);

    var funcName = "\_partResult" + funcNameSuffix + "\_";

    var resultFunc = this[funcName].bind(this, note, argObj\_);

    return priorities.map(resultFunc).filter(\_SATB.IS\_VALID\_RESULT);

};

$.\_partResultWithoutCache\_ = function(note, argObj\_, part) {

    return this.\_uncachedPartResult\_(note, argObj\_, part, "WithoutCache");

};

$.\_partResultWithCache\_ = function(note, argObj\_, part) {

    var cache = this.\_cache.partResult\_(note, argObj\_, part);

    if (\_SATB.IS\_VALID\_RESULT(cache)) return cache;

    return this.\_updatedPartResultWithCache\_(note, argObj\_, part);

};

$.\_updatedPartResultWithCache\_ = function(note, argObj\_, part) {

    var result =

            this.\_uncachedPartResult\_(note, argObj\_, part, "WithCache");

    this.\_cache.updatePartResult(note, argObj\_, part, result);

    return result;

};

$.\_uncachedPartResult\_ = function(note, argObj\_, part, funcNameSuffix) {

    var list = this["\_pairFuncListPart" + funcNameSuffix](note, part);

    if (list.length <= 0) return undefined;

return this.\_rules.chainedResult(list, note, argObj\_);

};

$.\_nonAssociativeResult = function(note, argObj\_, funcNameSuffix) {

    var list = this["\_pairFuncList" + funcNameSuffix](note);

    var defaultResult = this.\_pairs.default(note, argObj\_);

return this.\_rules.chainedResult(list, note, argObj\_, defaultResult);

};

$.\_pairFuncListWithoutCache = function(note) {

    return this.\_uncachedPairFuncList(note, "WithoutCache");

};

$.\_pairFuncListWithCache = function(note) {

    var cache = this.\_cache.pairFuncList\_(note);

    return cache || this.\_updatedPairFuncListWithCache(note);

};

$.\_updatedPairFuncListWithCache = function(note) {

    var list = this.\_uncachedPairFuncList(note, "WithCache");

    this.\_cache.updatePairFuncList(note, list);

    return list;

};

$.\_uncachedPairFuncList = function(note, funcNameSuffix) {

    var funcName = "\_pairFuncListPart" + funcNameSuffix;

    return this.\_rules.priorities(note).reduce(function(list, part) {

        return list.concat(this[funcName](note, part));

    }.bind(this), []);

};

$.\_pairFuncListPartWithCache = function(note, part) {

    var cache = this.\_cache.pairFuncListPart\_(note, part);

    return cache || this.\_updatedPairFuncListPartWithCache(note, part);

};

$.\_updatedPairFuncListPartWithCache = function(note, part) {

    var list = this.\_pairFuncListPartWithoutCache(note, part);

    this.\_cache.updatePairFuncListPart(note, part, list);

    return list;

};

$.\_pairFuncListPartWithoutCache = function(note, part) {

    var func = this.\_pairs.pairFuncs.bind(this.\_pairs, note);

    return this.\_cache.partListData(part, this.\_battler).map(func);

};

To be displayed as something like this:

$.result = function(note, argObj\_) {

    if (!$gameSystem.satbParam("\_isCached")) {

        // $.\_uncachedResult START

        if (this.\_rules.isAssociative(note)) {

            // $.\_associativeResult START

                // $.\_partResults START

            var priorities = this.\_rules.priorities(note);

            var partResults = priorities.map(function(part) {

                    // $.\_partResultWithoutCache START

                        // $.\_uncachedPartResult\_ START

                            // $.\_pairFuncListPartWithoutCache START

                var func = this.\_pairs.pairFuncs.bind(this.\_pairs, note);

                var list = this.\_cache.partListData(

part, this.\_battler).map(func);

                            // $.\_pairFuncListPartWithoutCache END

                if (list.length <= 0) return undefined;

return this.\_rules.chainedResult(list, note, argObj\_);

                        // $.\_uncachedPartResult\_ END

                    // $.\_partResultWithoutCache END

            }).filter(\_SATB.IS\_VALID\_RESULT);

                // $.\_partResults END

            var defaultResult = this.\_pairs.default(note, argObj\_);

return this.\_rules.chainedResult(

partResults, note, argObj\_, defaultResult);

            // $.\_associativeResult START

        }

            // $.\_nonAssociativeResult START

                // $.\_pairFuncListWithoutCache START

                    // $.\_uncachedPairFuncList START

var priorities = this.\_rules.priorities(note);

        var list = priorities.reduce(function(list, part) {

                        // $.\_pairFuncListPartWithoutCache START

            var func = this.\_pairs.pairFuncs.bind(this.\_pairs, note);

            var l = this.\_cache.partListData(

part, this.\_battler).map(func);

                        // $.\_pairFuncListPartWithoutCache END

            return list.concat(l);

        }.bind(this), []);

                    // $.\_uncachedPairFuncList END

                // $.\_pairFuncListWithoutCache END

        var defaultResult = this.\_pairs.default(note, argObj\_);

return this.\_rules.chainedResult(

list, note, argObj\_, defaultResult);

            // $.\_nonAssociativeResult END

        // $.\_uncachedResult END

    }

    var cache = this.\_cache.result\_(note, argObj\_);

    if (\_SATB.IS\_VALID\_RESULT(cache)) return cache;

    // $.\_updatedResultWithCache START

        // $.\_uncachedResult START

    var result;

    if (this.\_rules.isAssociative(note)) {

            // $.\_associativeResult START

                // $.\_partResults START

        var priorities = this.\_rules.priorities(note);

        var partResults = priorities.map(function(part) {

                    // $.\_partResultWithCache START

            var cache = this.\_cache.partResult\_(note, argObj\_, part);

            if (\_SATB.IS\_VALID\_RESULT(cache)) return cache;

                        // $.\_updatedPartResultWithCache\_ START

                            // $.\_uncachedPartResult\_ START

                                // $.\_pairFuncListPartWithCache START

            var c = this.\_cache.pairFuncListPart\_(note, part);

            var list;

            if (c) {

                list = c;

            } else {

                                    // $.\_updatedPairFuncListPartWithCache START

                                        // $.\_uncachedPairFuncListPart START

                var func = this.\_pairs.pairFuncs.bind(this.\_pairs, note);

                list = this.\_cache.partListData(

part, this.\_battler).map(func);

                                        // $.\_uncachedPairFuncListPart END

                this.\_cache.updatePairFuncListPart(note, part, list);

                                    // $.\_updatedPairFuncListPartWithCache END

            }

                                // $.\_pairFuncListPartWithCache END

            var result = undefined;

            if (list.length > 0) {

result = this.\_rules.chainedResult(list, note, argObj\_);

            }

                            // $.\_uncachedPartResult\_ END

            this.\_cache.updatePartResult(note, argObj\_, part, result);

            return result;

                        // $.\_updatedPartResultWithCache\_ END

                    // $.\_partResultWithCache END

        }).filter(\_SATB.IS\_VALID\_RESULT);

                // $.\_partResults END

        var defaultResult = this.\_pairs.default(note, argObj\_);

result = this.\_rules.chainedResult(

partResults, note, argObj\_, defaultResult);

            // $.\_associativeResult START

    }

            // $.\_nonAssociativeResult START

                // $.\_pairFuncListWithCache START

    var cache = this.\_cache.pairFuncList\_(note), list;

    if (cache) {

        list = cache;

    } else {

                    // $.\_updatedPairFuncListWithCache START

                        // $.\_uncachedPairFuncList START

var priorities = this.\_rules.priorities(note);

        var list = priorities.reduce(function(list, part) {

                            // $.\_pairFuncListPartWithCache START

            var cache = this.\_cache.pairFuncListPart\_(note, part);

            var l;

            if (cache) {

                l = cache;

            } else {

                                // $.\_updatedPairFuncListPartWithCache START

                                    // $.\_uncachedPairFuncListPart START

                var func = this.\_pairs.pairFuncs.bind(this.\_pairs, note);

                l = this.\_cache.partListData(

part, this.\_battler).map(func);

                                    // $.\_uncachedPairFuncListPart END

                this.\_cache.updatePairFuncListPart(note, part, l);

                                // $.\_updatedPairFuncListPartWithCache END

            }

            return list.concat(l);

                // $.\_pairFuncListPartWithCache END

        }.bind(this), []);

                        // $.\_uncachedPairFuncList END

        this.\_cache.updatePairFuncList(note, list);

                    // $.\_updatedPairFuncListWithCache END

    }

                // $.\_pairFuncListWithCache END

    var defaultResult = this.\_pairs.default(note, argObj\_);

result = this.\_rules.chainedResult(list, note, argObj\_, defaultResult);

            // $.\_nonAssociativeResult END

        // $.\_uncachedResult END

    this.\_cache.updateResult(note, argObj\_, result);

    return result;

    // $.\_updatedResultWithCache END

};

Or this one without comments indicating the starts and ends of the inlined functions:

$.result = function(note, argObj\_) {

    if (!$gameSystem.satbParam("\_isCached")) {

        if (this.\_rules.isAssociative(note)) {

            var priorities = this.\_rules.priorities(note);

            var partResults = priorities.map(function(part) {

                var func = this.\_pairs.pairFuncs.bind(this.\_pairs, note);

                var list = this.\_cache.partListData(

part, this.\_battler).map(func);

                if (list.length <= 0) return undefined;

return this.\_rules.chainedResult(list, note, argObj\_);

            }).filter(\_SATB.IS\_VALID\_RESULT);

            var defaultResult = this.\_pairs.default(note, argObj\_);

return this.\_rules.chainedResult(

partResults, note, argObj\_, defaultResult);

        }

var priorities = this.\_rules.priorities(note);

        var list = priorities.reduce(function(list, part) {

            var func = this.\_pairs.pairFuncs.bind(this.\_pairs, note);

            var l = this.\_cache.partListData(

part, this.\_battler).map(func);

            return list.concat(l);

        }.bind(this), []);

        var defaultResult = this.\_pairs.default(note, argObj\_);

return this.\_rules.chainedResult(

list, note, argObj\_, defaultResult);

    }

    var cache = this.\_cache.result\_(note, argObj\_);

    if (\_SATB.IS\_VALID\_RESULT(cache)) return cache;

    var result;

    if (this.\_rules.isAssociative(note)) {

        var priorities = this.\_rules.priorities(note);

        var partResults = priorities.map(function(part) {

            var cache = this.\_cache.partResult\_(note, argObj\_, part);

            if (\_SATB.IS\_VALID\_RESULT(cache)) return cache;

            var c = this.\_cache.pairFuncListPart\_(note, part);

            var list;

            if (c) {

                list = c;

            } else {

                var func = this.\_pairs.pairFuncs.bind(this.\_pairs, note);

                list = this.\_cache.partListData(

part, this.\_battler).map(func);

                this.\_cache.updatePairFuncListPart(note, part, list);

            }

            var result = undefined;

            if (list.length > 0) {

result = this.\_rules.chainedResult(list, note, argObj\_);

            }

            this.\_cache.updatePartResult(note, argObj\_, part, result);

            return result;

        }).filter(\_SATB.IS\_VALID\_RESULT);

        var defaultResult = this.\_pairs.default(note, argObj\_);

result = this.\_rules.chainedResult(

partResults, note, argObj\_, defaultResult);

    }

    var cache = this.\_cache.pairFuncList\_(note), list;

    if (cache) {

        list = cache;

    } else {

var priorities = this.\_rules.priorities(note);

        var list = priorities.reduce(function(list, part) {

            var cache = this.\_cache.pairFuncListPart\_(note, part);

            var l;

            if (cache) {

                l = cache;

            } else {

                var func = this.\_pairs.pairFuncs.bind(this.\_pairs, note);

                l = this.\_cache.partListData(

part, this.\_battler).map(func);

                this.\_cache.updatePairFuncListPart(note, part, l);

            }

            return list.concat(l);

        }.bind(this), []);

        this.\_cache.updatePairFuncList(note, list);

    }

    var defaultResult = this.\_pairs.default(note, argObj\_);

result = this.\_rules.chainedResult(list, note, argObj\_, defaultResult);

    this.\_cache.updateResult(note, argObj\_, result);

    return result;

};

With just 1 click on $.result. Bear in mind that the actual codebase hasn't changed one bit, it's just that the IDE will display the codes from the original LIV form to the new HIV form. **The goal this feature's to keep the codebase in the LIV form, while still letting those who can handle HIV well to be able to read the codebase displayed in the HIV version.**

It's very unlikely for those who can only handle very LIV well to be able to fathom such a complicated and convoluted method with 73 lines and so many different [levels of varying abstractions](https://en.wikipedia.org/wiki/Abstraction_layer) and implementation details all mixed up together, not to mention the really vast amount of completely needless code duplication that aren't even easy nor simple to spot fast; Those who can handle very HIV well, however, may feel that a 73 line method is so small that they can hold everything inside in their head as a whole very quickly without a hassle.

Of course, one doesn't have to show everything at once, so besides the aforementioned feature that inlines everything in the reading mode with just 1 click, the IDE should also support inlining a function at a time. Let's say we're to reveal \_uncachedPairFuncListPart:

$.\_updatedPairFuncListPartWithCache = function(note, part) {

    var list = this.\_uncachedPairFuncListPart(note, part);

    this.\_cache.updatePairFuncListPart(note, part, list);

    return list;

};

Clicking that method name in the above method should lead to something like this:

$.\_updatedPairFuncListPartWithCache = function(note, part) {

// $.\_updatedPairFuncListPartWithCache START

var func = this.\_pairs.pairFuncs.bind(this.\_pairs, note);

    var list = this.\_cache.partListData(

part, this.\_battler).map(func);

// $.\_updatedPairFuncListPartWithCache END

    this.\_cache.updatePairFuncListPart(note, part, list);

    return list;

};

Similarly, clicking the method name updatePairFuncListPart should reveal the implemention details of that method of this.\_cache, provided that the IDE can access the code of that class.

Such an IDE, if even possible in the foreseeable future, should at least reduce the severity of traversing a deep call stack with tons of small functions for those who can handle very HIV well, if not removing the problem entirely, without forcing those who can only handle very LIV well to deal with HIV, and without the issue of fighting for refactoring in this regard.

**Summary**

In general, those who can handle very HIV well will prefer very long functions, as it'll be more effective and efficient to draw the full picture without missing any nontrivial relevant detail that way for such software engineers, while writing and reading very short functions are just going the opposite directions in their perspectives; Those who can only handle very LIV well will prefer very short functions, as it'll be easier and simpler to reason about well-defined abstractions(as long as they don't [leak](https://en.wikipedia.org/wiki/Leaky_abstraction) in nontrivial ways) that way for such software engineers, while writing and reading long functions are just going the opposite directions in their perspectives. **Ideally, we should be able to handle very HIV well while still being very tolerant towards LIV, so we'd be able to work well with codes having all kinds of information volume.** Unfortunately, very effective and efficient software engineers are generally very intolerant towards extreme ineffectiveness or inefficiencies(especially when those small function abstractions do [leak](https://en.wikipedia.org/wiki/Leaky_abstraction) in nontrivial ways), so all we can do is to try hard.

**Combining Information Density With Information Volume**

**Very HID + HIV =** [**Massive Ball**](https://en.wikipedia.org/wiki/Big_ball_of_mud) **Of Complicated And Convoluted** [**Spaghetti**](https://en.wikipedia.org/wiki/Spaghetti_code)[**Legacy**](https://en.wikipedia.org/wiki/Legacy_code)

Imagine that you're reading a well-documented 100k line function where almost every line's written like some of the most complex math formulae. I'd guess that even the best of the best software engineers will never ever want to touch this perverted beast again in their lives. Usually such codebase are considered dead and will thus be probably rewritten from scratch.

Of course, HID + HIV isn't always this extreme, as the aforementioned 73 line version of $.result also falls into this category. Even though it'd still be a hellish nightmare for most software engineers to work with if many functions in the codebase are written this way, it's still feasible to refactor them into very high quality code within a reasonably tight budget if we've the highest devotions, diligence and disciplines possible. While such an iron fist approach should only be the last resort, sometimes the it's called for so we should be ready.

Nevertheless, try to avoid HID + HIV as much as possible, unless the situation really, really calls for it, like optimizing a massive production codebase to death(e.g.: gameplay codes), or when the problem domain's so chaotic and unstable that no sane nor sensible architecture will survive for even just a short time(pathetic architectures can be way worse than none). If you still want to use this style even when it's clearly unnecessary, you should have the most solid reasons and evidence possible to prove that it's indeed doing more good than harm.

**Very HID + LIV = Otherwise High Quality Codes That Are Hard To Fathom At First**

For instance, the below codes falls into this category:

return isValid && (array || []).concat(object || canUseDefault && default);

Imagine that you're reading a codebase having mostly well-defined and well-documented small functions(but far from being mostly 1 liners) but most of those small functions are written like some the most complex math formulae. While fathoming such codes at first will be very difficult, because the functions are well-documented, those functions will be easy to edit once you've fathomed it with the help of those comments; Because the functions are small enough and well-defined, those functions will be easy to use once you've fathomed how they're being called with the help of those callers who're themselves high quality codes.

Of course, HID + LIV doesn't always mean small short term pains with large long term pleasures, as it's impossible to ensure that none of those abstractions will ever [leak](https://en.wikipedia.org/wiki/Leaky_abstraction) in nontrivial ways. While the codebase will be easy to work with when it only ever has bugs that are either caught by the test suite or have at least some obvious causes, such codebase can still be daunting to work with once it produces rare bugs that are hard to even reproduce, all because of the fact that it's very hard to form the full pictures with every last bit of nontrivial relevant detail of massive codebases having mostly small but very terse functions.

Nevertheless, as long as all things are kept in moderation(one can always try in this regard), HID + LIV is generally advantageous as long as the codebase's large enough to warrant large scale software architectures and designs(the lifespan of the codebase should also be long enough), but not so large that no one can form the full picture anymore, as the long term pleasures will likely be large and long enough to outweigh short term pains a lot here.

**Very LID + HIV = Excessively Verbose Codes With Tons Of Redundant Boilerplate**

Think of an extremely verbose codebase having full of boilerplate and exceptionally long functions. Maybe those functions are long because of the verbosity, but you usually can't tell before actually reading them all. Anyway, you'll probably feel that the codebase's just wasting lots of your time once you realize that most of those long functions aren't actually doing much. Think of the aforementioned 28 line verbose Javascript examples having an extremely easy, simple and small terse 1 line counterpart, and think of the former being ubiquitous in the codebase. I guess that even the most verbose software engineers will want to refactor it all, as working with it'd just be way too ineffective and inefficient otherwise.

Of course, LID + HIV isn't always that bad, especially when things are kept in moderation. At least, it'd be nice for most newcomers to fathom the codebase, so codebases written in this style can actually be very beginner-friendly, which is especially important for software teams having very high turnover rates. Even though it's unlikely to be able to work with such codebase effectively nor efficiently no matter how much you've fathomed it due to the heavy verbosity and loads of boilerplate, the problem will be less severe if it's short-lived. Also, writing codes in this style can be extremely fast at first, even though it'll gradually become slower and slower, so this style's very useful in at least prototyping/making PoCs.

Nevertheless, LID + HIV shouldn't be used on codebases that'd already be very large without the extra verbosity nor boilerplate, especially when it's going to have a very long life span. Just think of a codebase that can be controlled into the 100k scale all with very terse codes(but still readable), but reaching the 10M scale because of complete refactoring of all those terse codes into tons of verbose codes with boilerplate. Needless to say, almost no one will continue on this road if he/she knows that the codebase will become that large that way.

**Very LID + LIV = Too Many Small Functions With The Call Stacks Being Too Deep**

For instance, the below codes fall into this category:

/\* This is the original codes

$.\_chainedResult = function(list, note, argObj\_, initVal\_) {

    var chainedResultFunc = this.\_rules.chainResultFunc(note);

return chainedResultFunc(list, note, argObj\_, initVal\_);

};

\*/

// This is the refactored codes

$.\_chainedResult = function(list, note, argObj\_, initVal\_) {

    var chainedResultFunc = this.\_chainedResultFunc(note);

return this.\_runChainedResult(

list, note, argObj\_, initVal\_, chainedResultFunc);

};

$.\_chainedResultFunc = function(note) {

    return this.\_rules.chainResultFunc(note);

};

$.\_runChainedResult = function(list, note, argObj\_, initVal\_, resultFunc) {

return resultFunc(list, note, argObj\_, initVal\_);

};

//

Think of a codebase with less than 100k lines but with already way more than 1k classes/interfaces and 10k functions/methods. It's almost a given that the deepest call stack in the codebase will be so deep that it can even approach the 100 call mark. It's because the only way for very small functions to be very verbose with tons of boilerplate is that most of those small functions aren't actually doing anything meaningful. We're talking about deeply nested [delegates](https://en.wikipedia.org/wiki/Delegation_(object-oriented_programming))/[forwarding](https://en.wikipedia.org/wiki/Forwarding_(object-oriented_programming)) functions which are all indeed doing very easy, simple and small jobs, and tons of interfaces or explicit dependencies having only 1 implementation or concrete [dependency](https://en.wikipedia.org/wiki/Dependency_injection)(configurable options with only 1 option ever used also has this issue).

Of course, LID + LIV does have its places, especially when the business requirements always change so abruptly, frequently and unpredicably that even the most reasonable assumptions can be suddenly violated without any reason at all(I've worked with 1 such project). As long as there can still be sane and sensible architectures that can last very long, if the codebase isn't flexible in almost every direction, the software teams won't be able to make it when they've to implement absurd changes with ridiculously tight budgets and schedules. And the only way for the codebase to be possible to be so flexible is to have as many well-defined interfaces and seams as possible, as long as everything else is still in moderation. For the newcomers, the codebase will seem to be [overengineered](https://en.wikipedia.org/wiki/Overengineering) over nothing already happened, but that's what you'd likely do when you can never know what's invariant.

Nevertheless, LID + LIV should still be refactored once there are solid reasons and evidences to prove that the codebase can begin to stablize, or the hidden [technical debt](https://en.wikipedia.org/wiki/Technical_debt) incurred from excessive [overengineering](https://en.wikipedia.org/wiki/Overengineering) can quickly accumulate to the point of no return. At that point, even understanding the most common call stack can be almost impossible. Of course, if the codebase can really never stablize, then one can only hope for the best and be prepared for the worst, as such projects are likely [death marches](https://en.wikipedia.org/wiki/Death_march_(project_management)), or slowly becoming one. Rare exceptions are that, some codebases have to be this way, like the default RPG Maker MV codebase, due to the business model that any RPG Maker MV user can have any feature request and any RPG Maker MV plugin developer can develop any plugin with any feature.

**Summary**

**While information density and volume are closely related, there's no strict implications from one to the other, meaning that there are different combinations of these 2 factors and the resultant style can be very different from each other.** For instance, HID doesn't imply LIV nor vice versa, as it's possible to write a very terse long function and a very verbose short function; LID doesn't imply HIV nor vice versa for the very same reasons. In general, the following largely applies to most codebases, even when there are exceptions:

**Very HID + HIV =** [**Massive Ball**](https://en.wikipedia.org/wiki/Big_ball_of_mud) **Of Complicated And Convoluted** [**Spaghetti**](https://en.wikipedia.org/wiki/Spaghetti_code)[**Legacy**](https://en.wikipedia.org/wiki/Legacy_code)

**Very HID + LIV = Otherwise High Quality Codes That Are Hard To Fathom At First**

**Very LID + HIV = Excessively Verbose Codes With Tons Of Redundant Boilerplate**

**Very LID + LIV = Too Many Small Functions With The Call Stacks Being Too Deep**

**Teams With Programmers Having Different Styles**

**Very HID/HIV + HID/LIV = Too Little Architecture vs Too Weak To Fathom Codes**

While both can work with very HID well, their different capacities and takes on information volume can still cause them to have ongoing significant conflicts. The latter values codebase quality over software engineer mental capacity due to their limits on taking information volume, while the former values the opposite due to their exceptionally strong mental power.

Thus the former will likely think of the latter as being too weak to fathom the codes and they're thus the ones to blame, while the latter will probably think of the former as having too little architecture in mind and they're thus the ones to blame, as architectures that are beneficial or even necessary for the latter will probably be severe obstacles for the former.

**Very HID/HIV + LID/HIV = Being Way Too Complex vs Doing Too Little Things**

While both can work with very HIV well, their different capacities and takes on information density can still cause them to have ongoing significant conflicts. The latter values function simplicity over function capabilities due to their limits on taking information density, while the former values the opposite due to their extremely strong information density decoding.

Thus the former will likely think of the latter as doing too little things that actually matter in terms of important business logic as simplicity for the latter means time wasted for the former, while the latter will probably think of the former as being too needlessly complex when it comes to implementing important business logic, as development speed for the former means complexity that are just too high for the latter(no matter how hard they try).

**Very HID/HIV + LID/LIV =** [**Over-Optimization**](https://en.wikipedia.org/wiki/Program_optimization#When_to_optimize) **Freak vs** [**Over-Engineering**](https://en.wikipedia.org/wiki/Overengineering) **Freak**

It's clear that these 2 groups are at the complete opposites - The former preferring [massive balls](https://en.wikipedia.org/wiki/Big_ball_of_mud) of complicated and convoluted [spaghetti](https://en.wikipedia.org/wiki/Spaghetti_code) [legacy](https://en.wikipedia.org/wiki/Legacy_code) over too many small functions with the call stacks being too deep due to the heavy need of optimizing the codebase to death, while the latter preferring the opposite due to the heavy need of making the codebase very flexible.

Thus the former will likely think of the latter as [over-engineering](https://en.wikipedia.org/wiki/Overengineering) freaks while the latter will probably think of the former as [over-optimization](https://en.wikipedia.org/wiki/Program_optimization#When_to_optimize) freaks, as codebase optimization and flexibility are often somehow at odds with each other, especially when one is heavily done.

**Very HID/LIV + LID/HIV = Too Concise/Organized vs Too Messy/Verbose**

It's clear that these 2 groups are at the complete opposites - The former preferring otherwise high quality codes that are hard to fathom at first over excessively verbose codes with tons of redundant boilerplate due to the heavy emphasis on the large long term pleasures, while the latter preferring the opposite due to the heavy emphasis on the small short term pains.

Thus the former will likely think of the latter as being too messy and verbose while the latter will probably think of the former as being too concise and organized, as long term pleasures from the high codebase qualities are often at odds with short term pains of the newcomers fathoming the codebase at first, especially when one is heavily emphasized over the other.

**Very HID/LIV + LID/LIV = Too Hard To Read At First vs Too Ineffective/Inefficient**

While both can only work with very LIV well, their different capacities and takes on information density can still cause them to have ongoing significant conflicts. The latter values the learning cost over maintenance cost(the cost of reading already fathomed codes during maintenance) due to their limits on taking information density, while the former values the opposite due to their good information density skill and reading speed demands.

Thus the former will likely think of the latter as being too ineffective and inefficient when writing codes that are easy to fathom in the short term but time-consuming to read in the long term, while the latter will likely think of the former as being too harsh to newcomers when writing codes that are fast to read in the long term but hard to fathom in the short term.

**Very LID/HIV + LID/LIV = Too Beginner Friendly vs Too Flexible For Impossibles**

While both can only work with very LID well, their different capacities and takes on information volume can still cause them to have ongoing significant conflicts. The former values codebase beginner friendliness over software flexibility due to their generally lower tolerance on very small functions, while the latter values the opposite due to their limited information volume capacity and high familiarity with very small and flexible functions.

Thus the former will likely think of the latter as being too flexible towards cases that are almost impossible to happen under the current business requirements due to such codebases being generally harder for newcomers to fathom, while the latter will likely think of the former as being too friendly towards beginners at the expense of writing too rigid codes due to codebases being beginner friendly are usually those just thinking about the present needs.

**Summary**

**It seems to me that many coding standard/style conflicts can be somehow explained by the conflicts between HID and LID, and those between HIV and LIV, especially when both sides are being more and more extreme.** The combinations of these conflicts may be:

**Very HID/HIV + HID/LIV = Too Little Architecture vs Too Weak To Fathom Codes**

**Very HID/HIV + LID/HIV = Being Way Too Complex vs Doing Too Little Things**

**Very HID/HIV + LID/LIV =** [**Over-Optimization**](https://en.wikipedia.org/wiki/Program_optimization#When_to_optimize) **Freak vs** [**Over-Engineering**](https://en.wikipedia.org/wiki/Overengineering) **Freak**

**Very HID/LIV + LID/HIV = Too Concise/Organized vs Too Messy/Verbose**

**Very HID/LIV + LID/LIV = Too Hard To Read At First vs Too Ineffective/Inefficient**

**Very LID/HIV + LID/LIV = Too Beginner Friendly vs Too Flexible For Impossibles**

**Conclusions**

Of course, one doesn't have to go for the HID, LID, HIV or LIV extremes, as there's quite some middle grounds to play with. In fact, **I think the best of the best software engineers should deal with all these extremes well while still being able to play with the middle grounds well, provided that such an exceptional software engineer can even exist at all.** Nevertheless, it's rather common to work with at least some of the software engineers falling into at least 1 extremes, so we should still know how to work well with them. After all, nowadays most of the real life business codebase are about teamwork but not lone wolves.

By exploring the importance of information density, information volume and their relationships, I hope that this article can help us think of some aspects behind codebase readability and the nature of conflicts about it, and that we can be more able to deal with more different kinds of codebase and software engineers better. I think that it's more feasible for us to be able to read codebase with different information density and volume than asking others and the codebase to accommodate with our information density/volume limitations.

Also, this article actually implies that readability's probably a complicated and convoluted concept, as it's partially objective at large(e.g.: the existence of consistent formatting and meaningful naming) and partially subjective at large(e.g.: the ability to handle different kinds of information density and volume for different software engineers). **Maybe many avoidable conflicts involving readability stems from the tendency that many software engineers treat readability as easy, simple and small concept that are entirely objective.**