Path-Sensitive

SOFTWARE DESIGN QUIZ

Go beyond your peers in the next 5 minutes
Take our software design quiz now!

Should you split that file?

/ DECEMBER 01, 2023

You're a line programmer for EvilCorp, and it's just an average day working on some code to collapse the economy.

```
EvilCorp
TS collapse.ts ×
src > economics > TS collapse.ts > disruptSupplyChain
1   import {Product, type Country} from './types';
2
3
4   function collapseEconomy(country: Country) {
5       for (const product of country.exports) {
6           disruptSupplyChain(product);
7       }
8   }
9
10
```

Then you realize you need some code for disrupting supply chains.

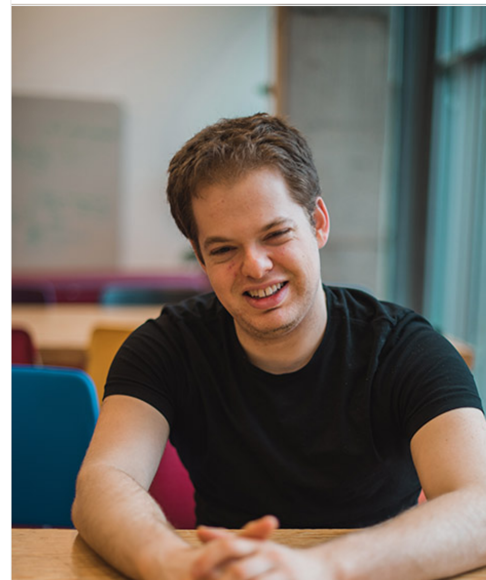
```
4   function disruptSupplyChain(product: Product) {
5       for (const input of product.inputs) {
6           const reducedAvailability =
7               estimateAvailabilityAfterDisruption(input);
8           const needed = amountNeeded(input, product);
9           if (reducedAvailability < needed) {
10              attackProduction(input);
11          }
12      }
13  }
```

Should you split it into a new file?

Let's say you do.

Pretty soon your directory looks like this:

About me



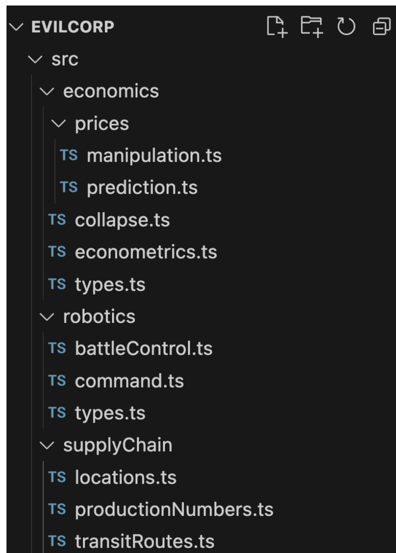
Jimmy Koppel

*Hello world! I'm Jimmy, and I help software engineers **learn to write better code**. Previously, I did my Ph. D. at MIT on ways to make program transformation and synthesis tools easier to build, a.k.a. "meta-metaprogramming." I blog mainly about improving code quality, and occasionally about life quality.*

Personal website

Labels

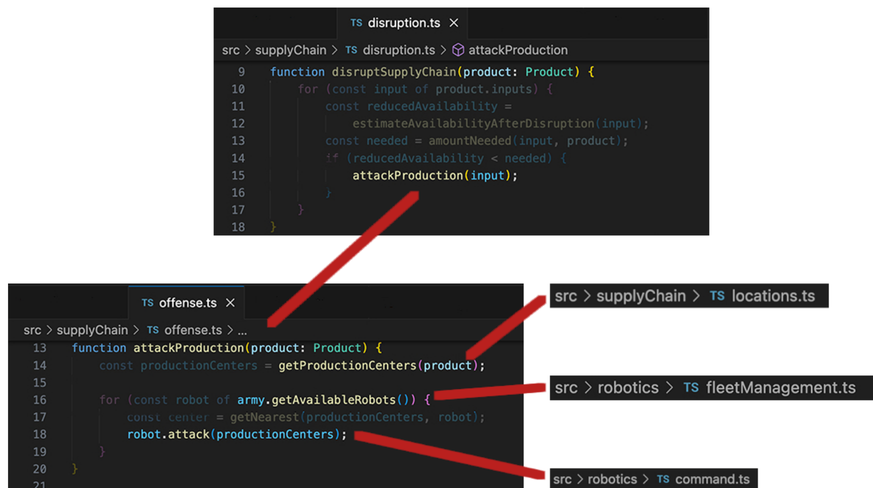
technical life-optimization
book-review researchy startups
talks fun personal tools



It's so well organized! You want to know what it tracks about the robot armies, it's right there.

Except that all your files look like this:

And the control flow through the files looks like this:

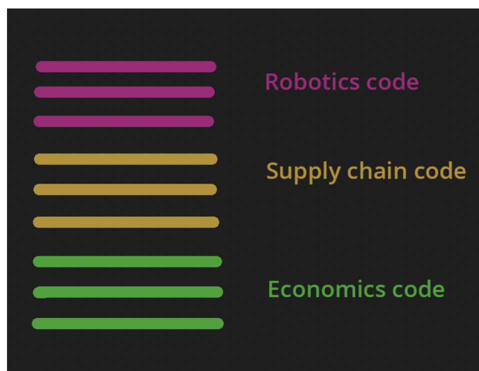


Now you're starting to regret having broken it up so aggressively.

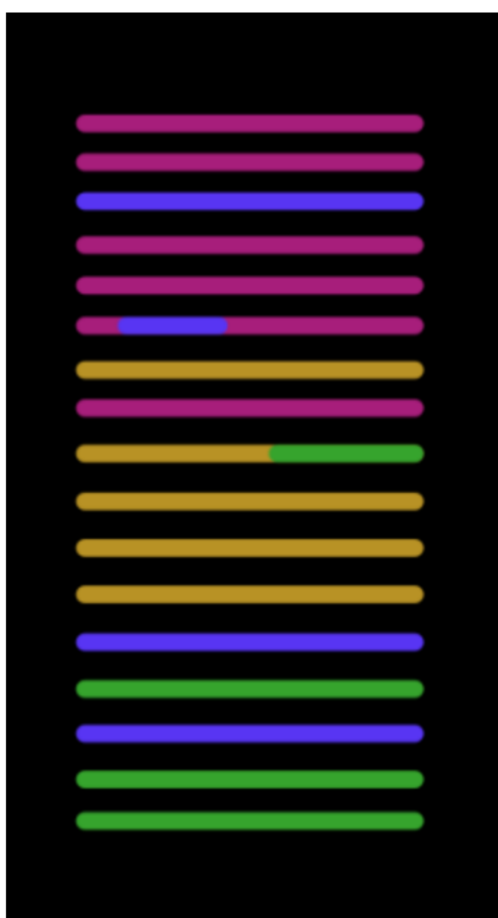
Now back up a bit.

Let's say you keep it in one file, at least for now.

Then you need to add some code for supply chain and robot info.

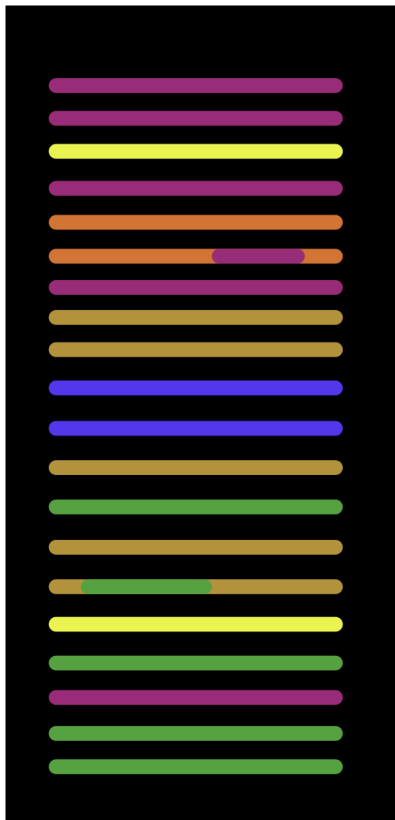


This repeats a few times. If you're lucky, the file looks like this, where lines of the same color represent related code:



There, we can see related code is mostly together, but there's some degradation, and a few things don't fit neatly into any category. Needs some weeding, but overall a decently-kept garden.

If you're less lucky, it looks more like this:



This is a file where there clearly used to be some structure, but now it's overgrown with chaos.

Of course, if you don't have the whole file committed in memory, it looks more like this:

Man, just figuring how it directs the army is a headache. That really should be broken out into its own file. But don't you wish you had done so earlier?

In nearly all ecosystems, programs consist of files consisting of text. When you break code into more files for more categories, it becomes easier to find and understand code for each category, but harder to read anything involving multiple categories. When you keep code together into fewer files, it becomes easier to track the control flow

for individual operations, but harder to form a mental map of the code.

What if I told you that you can eat the cake and have it too?

Here’s how.

The magic third way

Let’s look at your colleague Tom in the service division of the robotics department. He works on the repair manual that keeps the whole company’s army running smoothly. One day he’s working on the section for how to maintain the mirrors in the laser cannons.

Ch. 14: Maintenance of the Laser Cannon Arm

General schematics... 14-1
Accessing the mirror chamber.. 14-2
Inspecting the mirror for cracks 14-4
Polishing the mirror... 14-6

He realizes that he actually wants to add quite a few things about polishing the mirror. You see, the mirrors can only be polished with a custom nanoparticle solution, and so part of maintaining the mirror is really about maintaining the polish. Where to put this information?

Unlike in code, it’s a pretty big deal to “split stuff out into a new file,” since they like to keep everything in one volume for the technicians. Putting it in a new chapter would mean an awful lot of page flipping. And it’s quite messy to just mix in a lot of sections about maintaining the polish into the larger chapter on the laser cannon.

Ch. 14: Maintenance of the Laser Cannon Arm

General schematics... 14-1
Accessing the mirror chamber.. 14-2
Inspecting the mirror for cracks 14-4
Signs of an improperly-stored nanoparticle polish....................... 14-6
The Taub-Brown planarization test for polish degradation 14-7
Polishing the mirror... 14-10

But he has no problem adding them with more organization:

Ch. 14: Maintenance of the Laser Cannon Arm

14.1: General operations 14-1

General schematics 14-1

Accessing the mirror chamber 14-2

14.2: Maintaining the mirror 14-4

14.2.1: The nanoparticle polish 14-4

Signs of an improperly-stored nanoparticle polish 14-4

The Taub-Brown planarization test for polish degradation........... 14-5

14.2.2: Mirror repair 14-8

Inspecting the mirror for cracks 14-8

Polishing the mirror 14-10

That's the normal way to organize books, with chapters and subchapters (and sub-subchapters). Or, in HTML: h1, h2, etc.

We have them in code too.

They look like this:

```
/***************************************************
 **************** h1 in C/C++/Java/JS **************
 ***************************************************


/**************
 ********* This is an h2
 **************/



/********
 **** An h3
 ********/
```

Or this:

```
################ Python/Ruby/Bash H1 ###############

############## An H2

##### An H3
```

Or this:

```
----------------------------------------------------
--                    Haskell/Lua h1
----------------------------------------------------



-----------                An h2            -------



------ An h3
```

Or any of countless other variations. They all get the job done. I tend to like the ones that more visibly break up the text, so: like the first

bunch, except translated into whatever language I'm using.

I teach people a lot of things about software design. Some of them are things I dusted off from papers written in the 70's. Many more I can claim to have invented myself.

This one, not in the slightest. In fact, **here's** some CSS guys doing it, in the frontend framework "Semantic UI":

```
217   /********************************
218                Types
219   ********************************/
220
221   /*-------------------
222          Animated
223   --------------------*/
224
225   .ui.animated.button {
226     position: relative;
227     overflow: hidden;
228     padding-right: 0em !important;
229     vertical-align: @animatedVerticalAlign;
230     z-index: @animatedZIndex;
231   }
```

And here's some smart-contract developers:

```
107     // ====================================
108     // =========== Modifier ===============
109     // ====================================
110
111     // Both modifiers below are using OpenZeppelin's AccessControl.sol with custom roles under the hood
112
113     /// @notice Reverts UNAUTHORIZED() if the caller is not a pool manager
114     /// @param _poolId The pool id
115     modifier onlyPoolManager(uint256 _poolId) {
116         _checkOnlyPoolManager(_poolId);
117         _;
118     }
119
120     /// @notice Reverts UNAUTHORIZED() if the caller is not a pool admin
121     /// @param _poolId The pool id
122     modifier onlyPoolAdmin(uint256 _poolId) {
123         _checkOnlyPoolAdmin(_poolId);
124         _;
125     }
126
127     //  ====================================
128     //  ==== External/Public Functions =====
129     //  ====================================
130
```

But I can say that I've never seen anyone else write about it explicitly, nor take them as far as I do.

"Jimmy," one listener commented. "At my workplace, I've seen a lot of code with these sections, but stuff keeps getting added in the middle, and then the sections become meaningless."

"Is it too hard to just add a subsection for exactly the stuff added?"

"Actually" he replied, "I don't think I've ever seen subsections."

But my code these days has it everywhere. On occasion to three or more levels of organization.

```
81
82  ################################################################
83  ###################### Identifier clusters ####################
84  ################################################################
85
86  #######################
87  ########## Type
88  #######################
89
90  ########
91  ### Abstraction barrier around identifiers
92  ########
93
94  Identifier = str
95
96  def identifier_to_json(x: Identifier) -> str:
97      return x
```

(What in the world is this code doing, just renaming strings? Another deep idea and another discussion. Short version: Trying to get the benefits of having a fancy datatype for identifiers without actually doing any work.)

Aggressively splitting files into sections and (sub-)subsections is the biggest way my code has changed in the last 5 years. It requires little skill, and, once you build the habit, little effort. But I've found it makes a huge difference in how pleasant it is to live in a codebase.

Cognitive load and design reconstruction

Hopefully I don't have to argue too hard that code organized into sections and subsections is nicer to read, if not to write. Here are two versions of a code snippet (**source**) from Semantic UI: the original, and one with the section dividers removed. Personally, even at a glance, I find the version with them present more inviting.

```
/*******************************
             States
*******************************/

/*--------------
      Hover
---------------*/

.ui.button:hover {
  background-color: @hoverBackgroundColor;
  background-image: @hoverBackgroundImage;
  box-shadow: @hoverBoxShadow;
  color: @hoverColor;
}

.ui.button:hover .icon {
  opacity: @iconHoverOpacity;
}

/*--------------
      Focus
---------------*/

.ui.button:focus {
  background-color: @focusBackgroundColor;
  color: @focusColor;
  background-image: @focusBackgroundImage !important;
  box-shadow: @focusBoxShadow !important;
}

.ui.button:focus .icon {
  opacity: @iconFocusOpacity;
}

/*--------------
      Down
---------------*/

.ui.button:active,
.ui.active.button:active {
  background-color: @downBackgroundColor;
  background-image: @downBackgroundImage;
  color: @downColor;
  box-shadow: @downBoxShadow;
}

/*--------------
     Active
---------------*/

.ui.active.button {
  background-color: @activeBackgroundColor;
  background-image: @activeBackgroundImage;
  box-shadow: @activeBoxShadow;
  color: @activeColor;
}
.ui.active.button:hover {
  background-color: @activeHoverBackgroundColor;
  background-image: @activeHoverBackgroundImage;
  color: @activeHoverColor;
}
.ui.active.button:active {
  background-color: @activeBackgroundColor;
  background-image: @activeBackgroundImage;
}
```

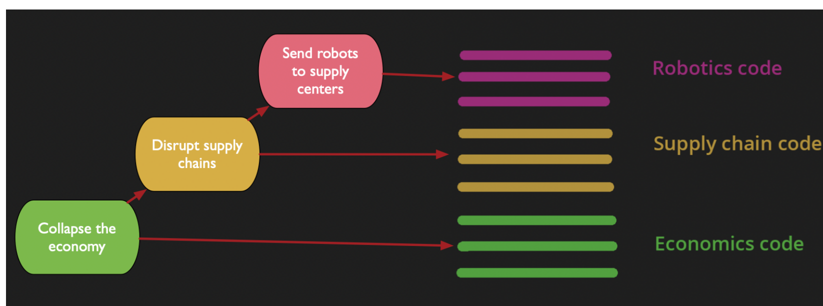
```
.ui.button:hover {
  background-color: @hoverBackgroundColor;
  background-image: @hoverBackgroundImage;
  box-shadow: @hoverBoxShadow;
  color: @hoverColor;
}

.ui.button:hover .icon {
  opacity: @iconHoverOpacity;
}

.ui.button:focus {
  background-color: @focusBackgroundColor;
  color: @focusColor;
  background-image: @focusBackgroundImage !important;
  box-shadow: @focusBoxShadow !important;
}

.ui.button:focus .icon {
  opacity: @iconFocusOpacity;
}

.ui.button:active,
.ui.active.button:active {
  background-color: @downBackgroundColor;
  background-image: @downBackgroundImage;
  color: @downColor;
  box-shadow: @downBoxShadow;
}

.ui.active.button {
  background-color: @activeBackgroundColor;
  background-image: @activeBackgroundImage;
  box-shadow: @activeBoxShadow;
  color: @activeColor;
}
.ui.active.button:hover {
  background-color: @activeHoverBackgroundColor;
  background-image: @activeHoverBackgroundImage;
  color: @activeHoverColor;
}
.ui.active.button:active {
  background-color: @activeBackgroundColor;
  background-image: @activeBackgroundImage;
}
```

There are actually some pretty deep reasons why sub-file organization works.

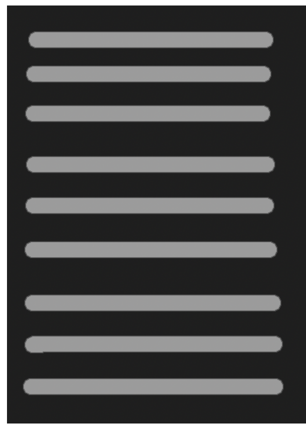
We saw that splitting a file comes with advantages and disadvantages. As does not splitting.

But, actually, for not splitting, all the disadvantages were in reading it.

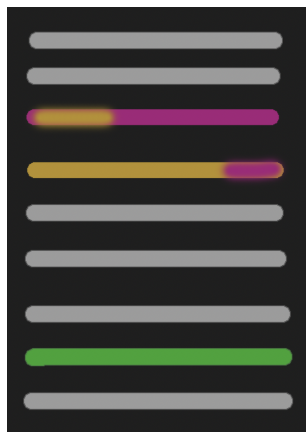
When you start a new feature, you have some high-level intentions. By some process, you turn the high-level intentions into low-level intentions into code.



But when someone first looks at a file, it's just a blob.



Then they start to read and build an understanding of each piece.



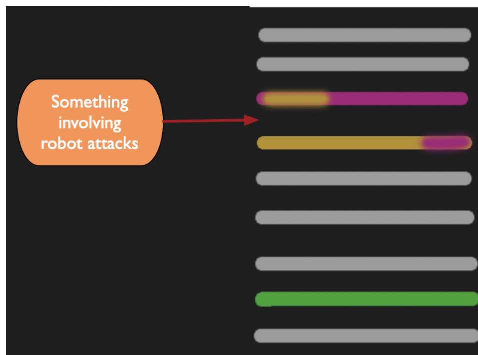
As they understand more pieces, they can begin to understand how they fit together into a bigger picture.

But all this is wasted work! The reader is just trying to reconstruct knowledge that was already known to the writer!

That's a very general problem. Anything done to counter it falls under the umbrella of what I call the **Embedded Design Principle**. Splitting a file into sections is just one particularly effective instance of this broader idea. As poetically explained in **The 11 Aspects of Good Code**:

> ***Good code makes it easy to recover the intent of the programmer***
>
> *A programmer dreams a new entity. Her mind gradually turns dream into mechanism, mechanism into code, and the dreamed entity is given life.*
>
> *A new programmer walks in and sees only code. But in his mind, as he reads and understands, the patterns emerge. In his mind, code shapes itself into mechanism, and mechanism shapes itself into dream. Only then can he work. For in truth, a modification to the code is a modification to the dream.*
>
> *Much of a programmer's work is in recovering information that was already present in the mind of the creator. It is thus the creator's job to make this as simple as possible.*

Back to the robot armies. The reader has started to piece together a bigger picture.



In this example, the code was written in three sections and then not edited. That brings an offer: understand the first three functions, and you understand the big ideas of the mechanics behind sending forth a robot army. Understand the next three, and you understand the bigger picture. But the reader in the picture hasn't found that structure yet.

Piecing this code together is like a jigsaw puzzle. And in a jigsaw puzzle, if I were to give you a box with only pieces from the left half, and a box with only the pieces from the right half, it would be more than twice as easy.[1] That's a lot like what you're doing for the reader by labeling code sections.[2]

There's one more benefit too. I and many others I showed this to report a sense of relaxation and calm from skimming through a well-sectioned file, a lot like coming home to a clean room. I think what's going on here is cognitive ease: there's a psychological phenomenon in which easy things literally cause happiness. There's an entire chapter on it in Kahnemann's Thinking Fast and Slow.

Oh, and then there's also how naming is one of the two hard problems of computer science (the others being cache invalidation and off-by-one errors). If you put two functions that coordinate robotics surrounding and invading a factory into a file, then you're going to want to think of some general name that captures both these and everything similar that should go in the same file. That sounds kinda tough; my best is "offensive_tactics.ts." But you just cordon these off into a little section of a larger file containing the whole supply-chain disruption logic, then naming that section is a much lower bar. After you find yourself writing additional related functions, then you can break it off into a new file as easily as you can change a subchapter in a book into a full chapter.

So there's a lot of costs and benefits to breaking up a file vs. keeping it together, and we've seen that having sections and subsections does a

lot to lower the cost of keeping it together. But actually, it's pretty rare that I've seen people go too aggressive in breaking up files. More often I see people who think breaking up a file would make it more organized, but there's just too much inertia. And the bigger a file grows, the harder it becomes to break out meaningful components.

That's why having a handful of giant files used to be the hallmark of a bad codebase, one completely disorganized. But this is the real greatness of sections: it's a way to get much of the benefits of splitting up files, but it feels more like jotting down a thought you had than actually doing work. And if you keep things organized in sections, then it's not any harder to break apart a file later than it is now.

So now we know that, just by recording a little bit more of your thinking when writing code, it's possible to have files which are both large and well-organized. And doing so lets you read code faster, follow control-flow better, delay having to find good names, and literally injects happiness into your life. Let's make our files large again!

Of course, this is still not the easiest thing you can do to lower the cost of large files.

That would be buying a bigger monitor.

Thank you to Jonathan Camenisch, James He, and Supachai "Champ" Suwanthip for discussion on the ideas behind this blog post.

Thank you to Benoît Fleury, Torbjörn Gannholm, Oliver Chambers, and William Berglund for comments on earlier drafts.

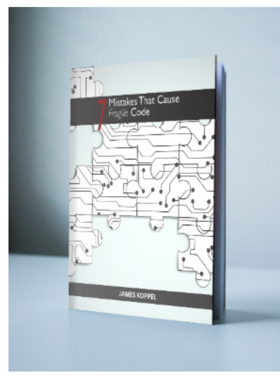
[1] *I had to check this one and, **it turns out**, average solving time for jigsaws is remarkably linear in the number of pieces. But, if you like jigsaws and know some computer science, we can reason about the complexity of each step of solving: first find the corners and edges (linear), then group pieces by region (linear-ish), then solve the parts of the puzzle where each piece looks distinct (linear to quadratic), then solve the parts of the puzzle where the pieces all look similar (near quadratic). Through this lens, a large jigsaw is actually composed of many subregions, each of which could take near-quadratic solving time. In the worst case, the jigsaw is just a solid color, and you're stuck comparing each edge pairwise, which is clearly quadratic unless you're really good at indexing on the shapes of the holes and protrusions. This invites the more accurate statement: for each of the quadratic-time subregions of a jigsaw puzzle, if I were to split the pieces into a left and right half, the solving speed for that subregion would roughly improve by 4x. This is both more accurate and a better metaphor for the effect of adding subdivisions to a source file.*

[2] *The ideal would be to do the code equivalent of handing someone a painting instead of cutting it up into jigsaw pieces in the first place. The programming equivalent would be to actually make your designs into the program. Choices for approaching that include writing with declarative libraries, using symbolic program synthesis techniques, or using ChatGPT and letting natural language be the code.*

TAGS:

Liked this post?

Join Arch-Engineer, the newsletter on how to write better code, and receive a free copy of the 7 Mistakes That Cause Fragile Code.



First name | Last name | Email-address | SUBSCRIBE

Related Articles

2 comments:

ANONYMOUS

Traditionally, source file sections were separated with ASCII FORM FEED characters. It's whitespace so (most) compilers ignore it.

Nicer text editors have convenient ways to operate on only the current "page", or navigate between pages, or narrow the view to only one page at a time. You can have it display the first comment on a page as a title. It's a pretty great system and I don't know why it's so rarely used.

Reply

Replies

JAMES KOPPEL

Wow, that's cool! Got any references?

Reply

Enter Comment

OLDER POST →

Copyright © 2015-18 James Koppel. All rights reserved.