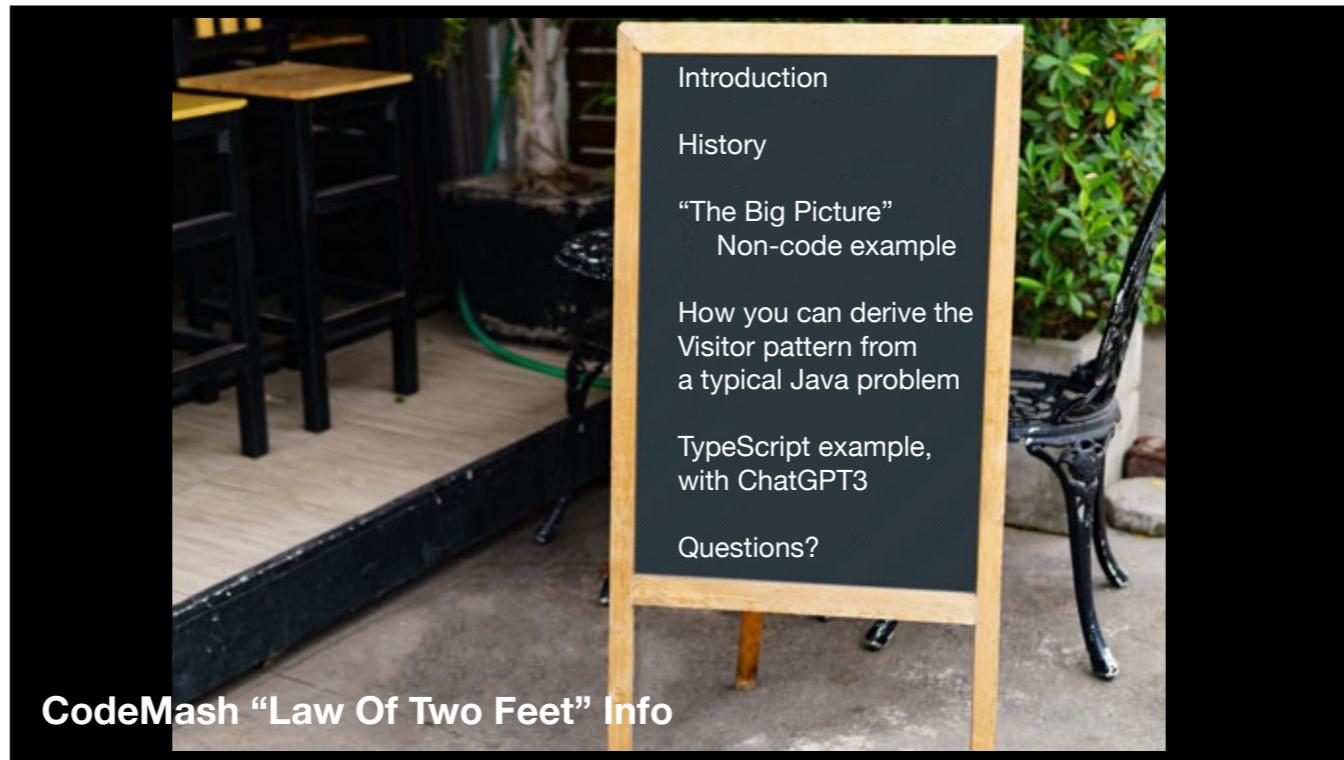




A Vibrant Visit With The Valuable And Versatile Visitor Design Pattern

Good afternoon! I'm Kelly Morrison, and this is a presentation on the Visitor design pattern. While we're waiting on people to filter in, let me explain this opening slide. When I was coming up with the title for this talk, the alliteration of the words starting with "V" made me think of the opening chapter of Alan Moore's "V for Vendetta", where the character called "V" has a speech using most of the "V" words in the dictionary. So I thought I'd throw "V" on the slide as a tip of the hat for providing the inspiration for this title.



CodeMash follows the “Law of Two Feet” rule, which means that if a presentation isn’t what you expected, or you feel like you’d get more information elsewhere, then you’re free to leave with no hard feelings. So, here’s a quick overview of what we’ll be covering today to inform your decision.

First, we’ll do a quick introduction so you know who I am. Then, we’ll look at a quick history of the Visitor design pattern. Next, I’ll explain it without using any computer terms - we’ll look at a “real world” example of what the Visitor design pattern is so you can get “The Big Picture.”

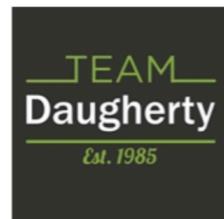
The bulk of the presentation will be devoted to showing a typical work problem that could be better implemented using the Visitor design pattern in Java, and we’ll see how it’s derived.

Finally, we’ll look at implementing the Visitor design pattern in TypeScript by using ChatGPT3, and I’ll show some of the things to look out for when taking this approach.

I’ll wrap up by entertaining any questions from the audience. All set? Okay - let’s go!

Kelly Morrison, Ph.D. Computer Engineering

Solution Architect - Daugherty Business Solutions



THE CLIMATE
CORPORATION



Marshall Space
Flight Center



PATHFIRE
Enabling Digital Media



The University of Georgia



AUBURN
UNIVERSITY



To give you a little background, my name is Kelly Morrison. I'm a consultant at Daugherty Business Solutions, which is a great company - we have a booth here at the conference if you'd like to come by and see what it takes to join us.

I have a bachelor's degree in mathematics from the University of Georgia (GO DAWGS!!!) and a doctorate from Auburn University. I've worked for many companies over the years - you can see a sampling of them here.



A long, long time ago...

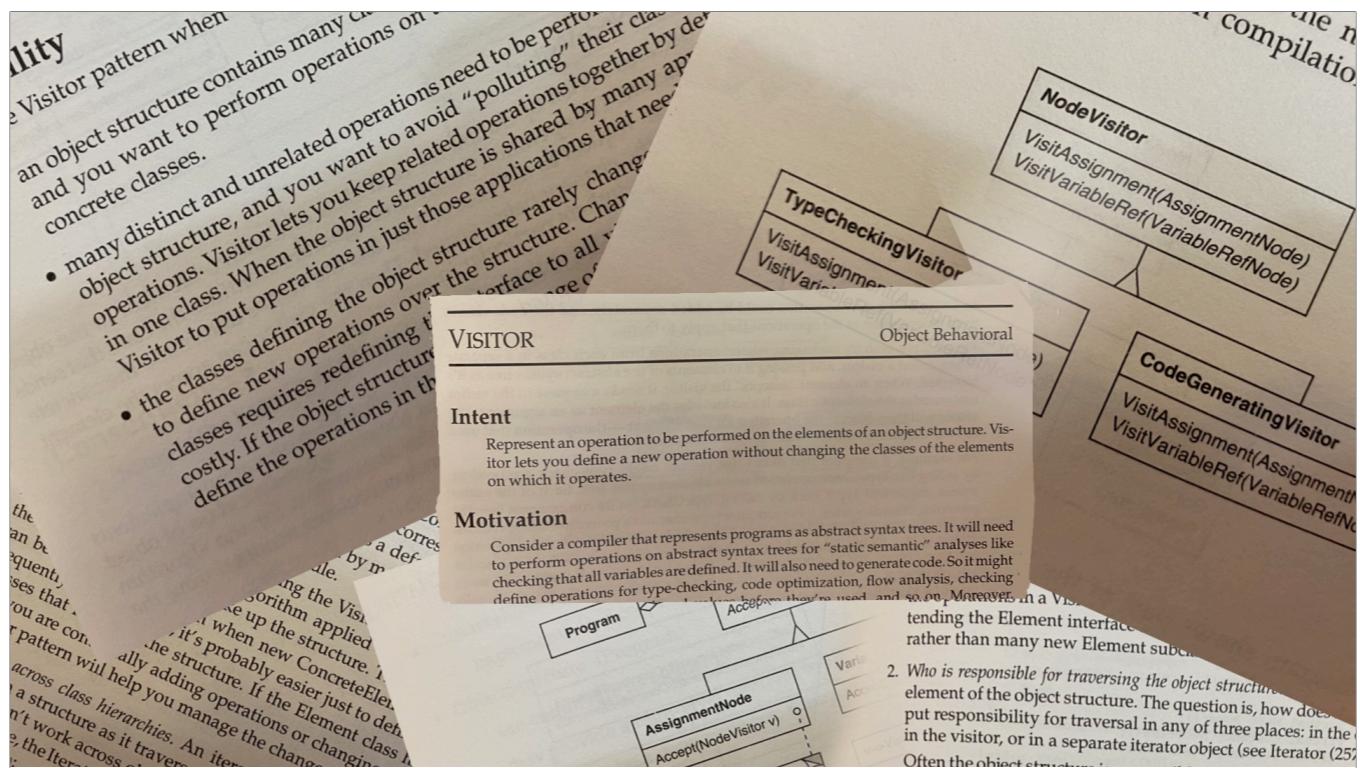
(1994)

To begin our presentation, I'd like to take you back in time to the distant year of ... 1994. A time when dinosaurs roamed the earth, Hootie and the Blowfish and Toad the Wet Sprocket were releasing albums, and "The Lion King" and Tom Cruise ruled the box office - so, in some ways, I guess it was similar to the present day. By the way, this is an actual Time magazine cover from 1994.



One of the most influential books in computer science was released in 1994: “Design Patterns”, by the “Gang of Four”. A few years later, a follow-up book was introduced called “Pattern Hatching”.

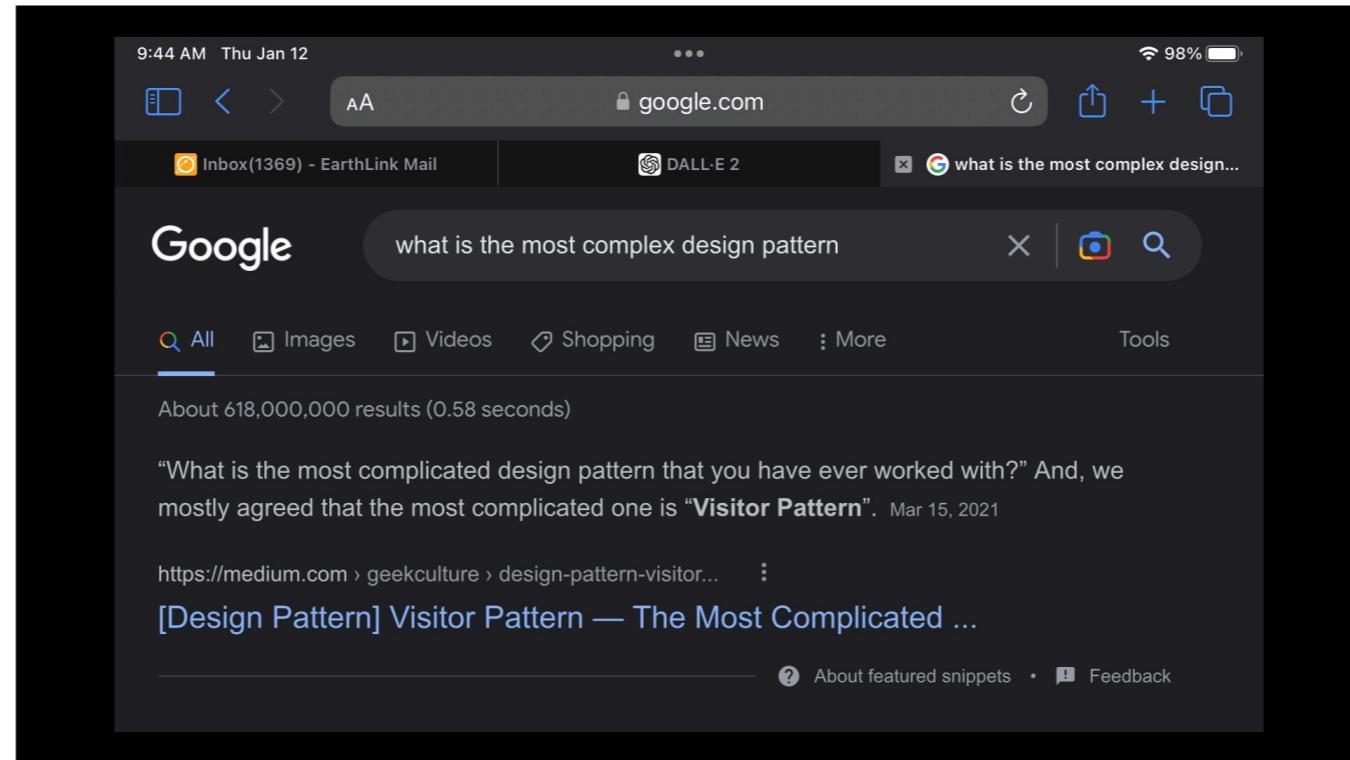
The “Design Patterns” book tried to introduce a standardized catalog of common patterns in software development. “Blueprints”, if you will. In recent years, it’s become trendy to downplay or disparage this book on Twitter, but the truth is that the patterns it introduced are still around. Many have been absorbed into frameworks or computer languages. Ever use a fluent API? That’s the Builder pattern. Ever used the Java IO library? It heavily uses the Decorator pattern. And on and on...



However, one of the patterns in the book is used infrequently: the “Visitor” design pattern. Many people regard it as one of the most complex and confusing patterns in the book. The original description is relatively vague, and the “motivation” given for it is to use it when writing a compiler, something that most software developers never do and have little understanding of.



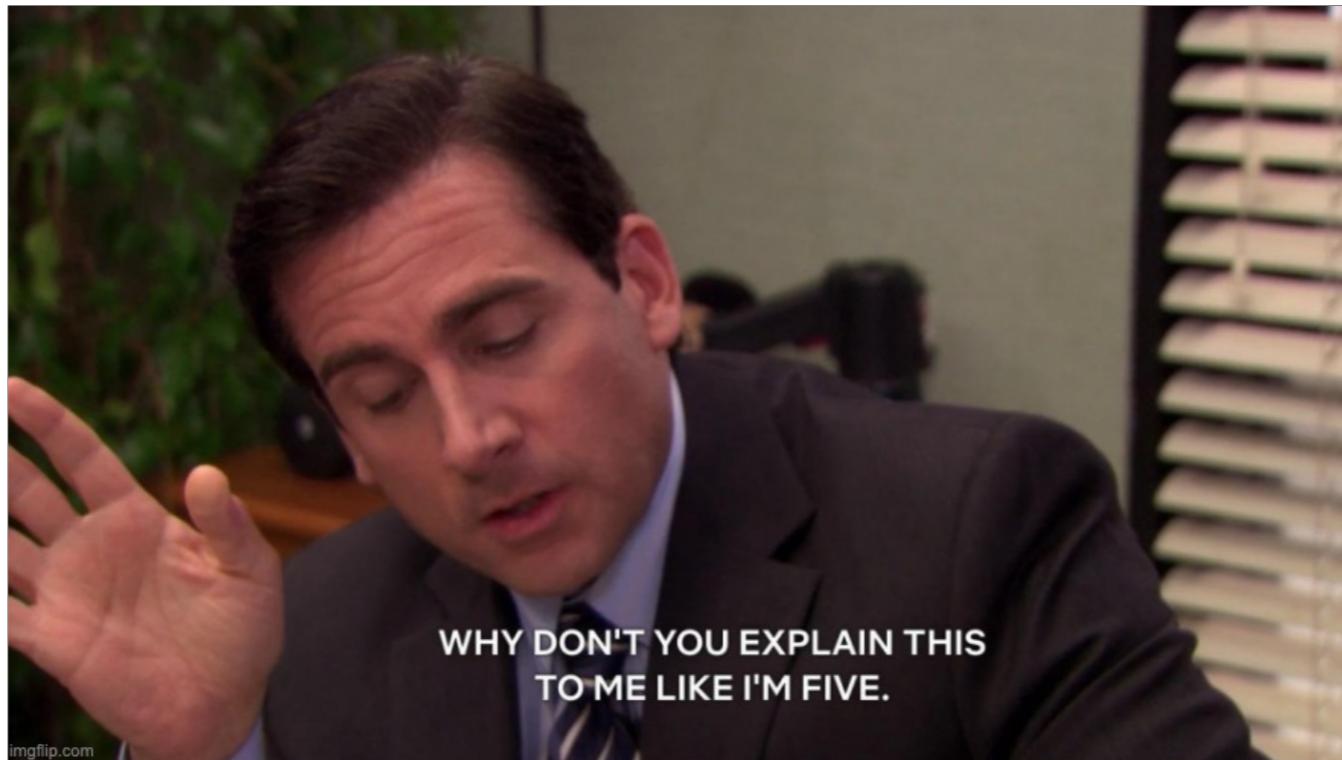
So - the pattern has a confusing description and an example using one of the more complicated problem spaces in computer science. Not surprisingly, many people skip past it.



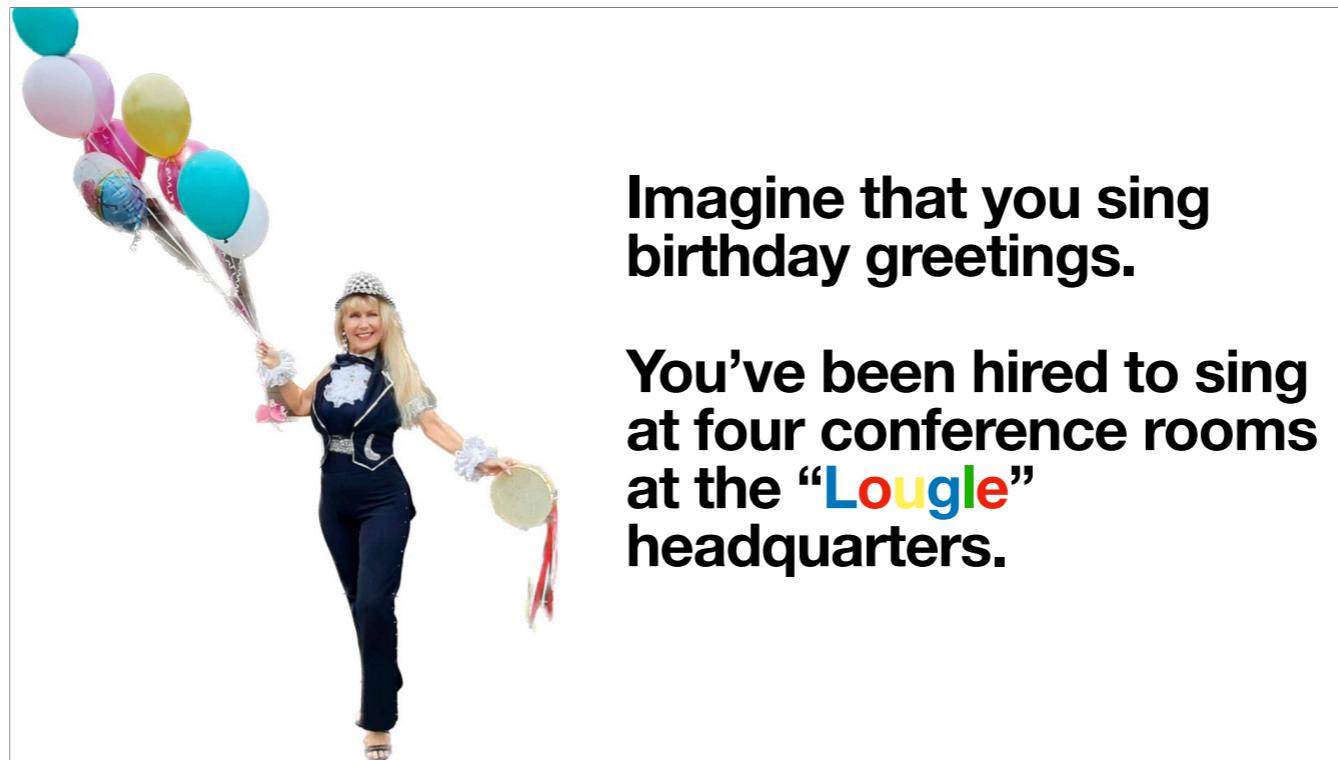
Just to show that things haven't changed, I Googled "what is the most complex design pattern" a few hours before this presentation... and, sure enough, the Visitor pattern was at the top of the results.



If the Visitor design pattern were the first pattern presented in the book, this would be many people's reaction.



The sad thing is that the Visitor design pattern is actually easy to understand, and it's incredibly useful in everyday work. My goal for today is to explain it to you using an ELI5 approach (Explain It Like I'm 5) so that you can understand its potential and recognize when it can be useful to you in everyday work.



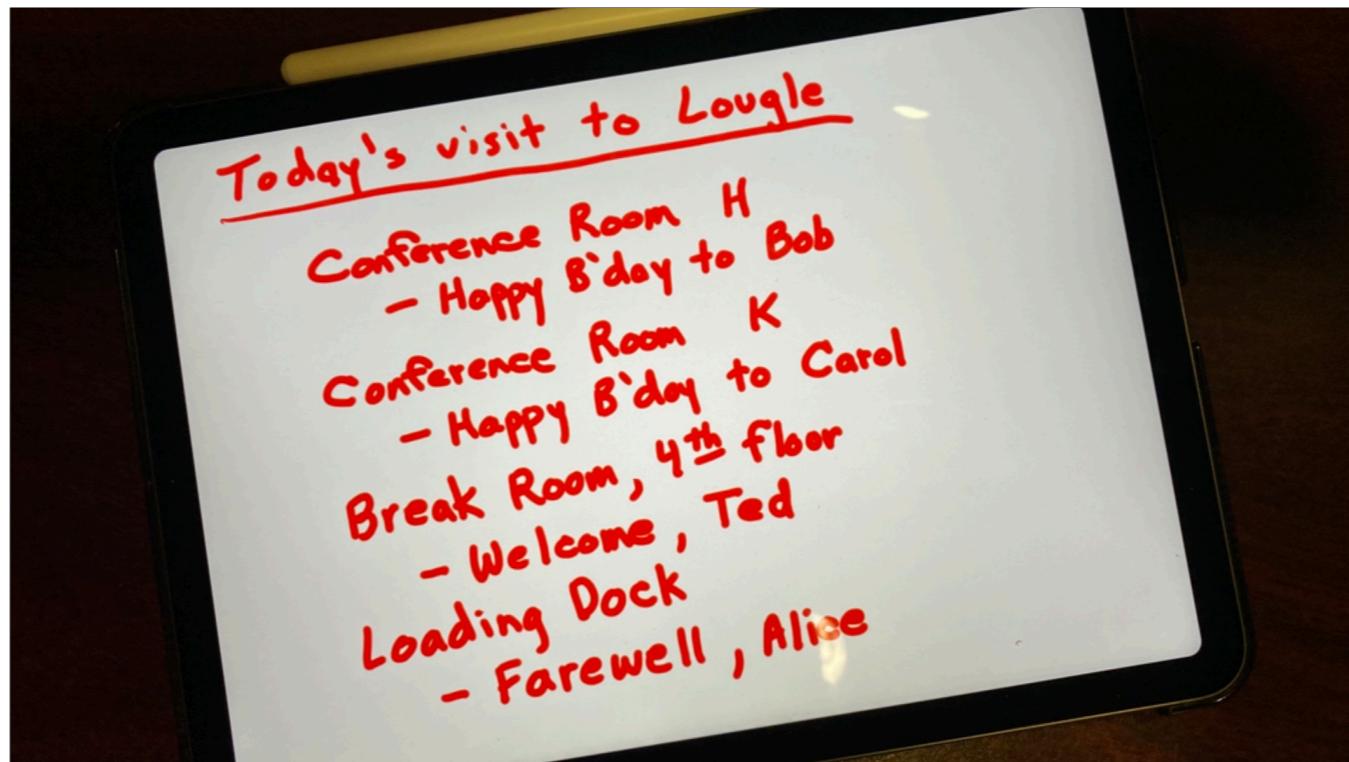
**Imagine that you sing
birthday greetings.**

**You've been hired to sing
at four conference rooms
at the “[Louple](#)”
headquarters.**

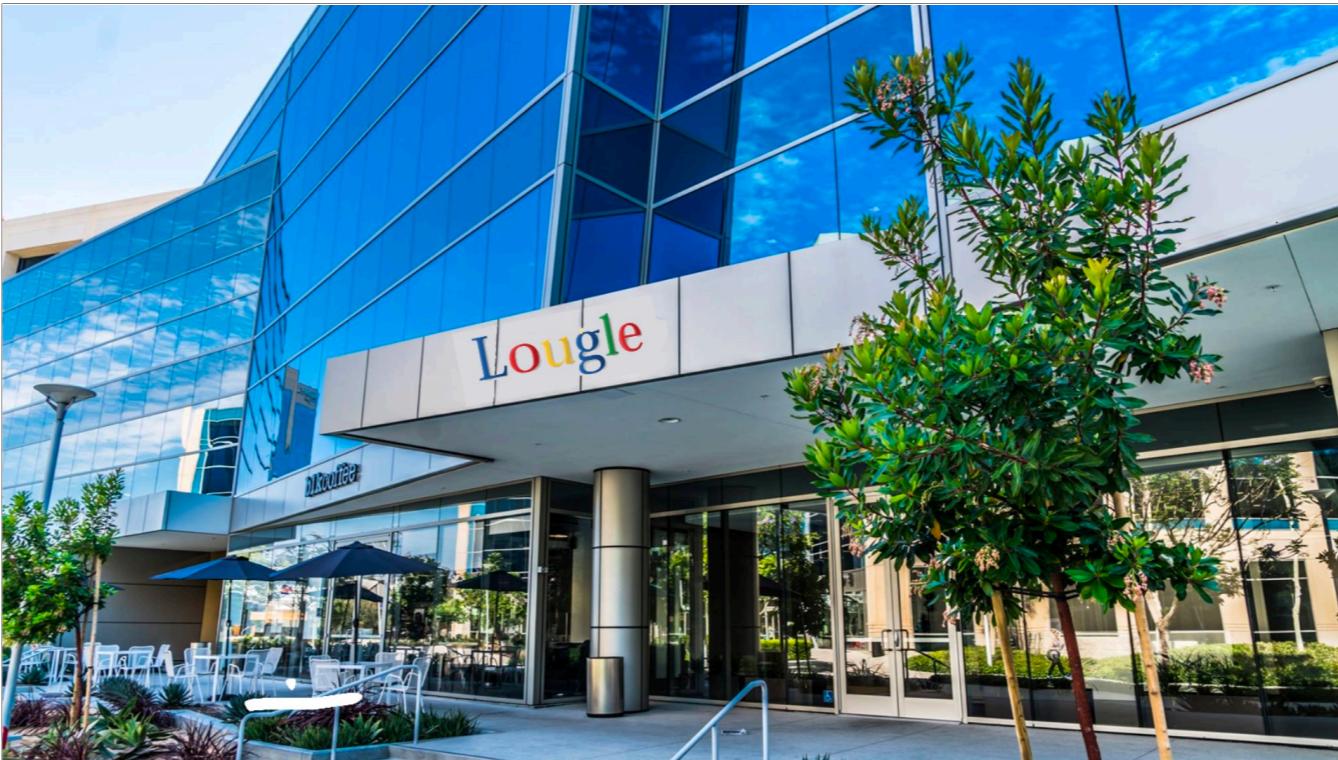
So, let's start by seeing how the Visitor design pattern works without using any code at all.

Let's imagine that we work for one of those companies that send people out to sing “happy birthday”. I love these people: they spend their entire work day making people happy, they're constantly visiting new places, they make some money, they never have to attend a morning stand-up, and they never have to waste time in a lengthy debate about Jira story points.

So yeah: let's pretend we're one of these people, and today we're going to sing at four different conference rooms at the “[Louple](#)” headquarters. (By the way, you'll notice I use a lot of memes and cultural references - feel free to shout them out). And remember this lady - you'll be seeing her a lot today.



Here's what your agenda might look like on your iPad. We're going to visit Lougle, and we have to sing to four different people at four different locations there. Easy enough, right?



So, we go over to Lougle headquarters, and the first thing we notice is “wow, that’s a big building. I hope I can find these people!”

“Welcome to Lougle!

We’re excited to have you here to sing “Happy Birthday” to Bob, Carol, Ted, and Alice.”



You go in the front door, and a strangely familiar intern welcomes you. (Remember him - we'll be seeing him frequently today). He tells you that everyone is excited to have you here to sing, and you're happy and feeling great, but then he does something that crushes your mood...



“Here’s a map of the building.

Have fun!”

He hands you this huge fold-out map of the company, says “here’s a map of the building”, then points to an escalator and says “have fun!”

You look at this complex, labyrinthine map, and you think “how am I ever going to find these people! This looks like anything BUT fun!”

This isn't good.

This isn't good. Why? Anyone in the audience want to hazard a guess?

What's wrong?

- We shouldn't force a casual visitor to have to figure out where to go in an enormous building
- Security risk - what if the visitor wanders into a secure, or confidential, or dangerous area?
- What if the appointment changed? Maybe Carol's party got moved to a different room because the air conditioning was broken in Conference Room H?

Great. So, here are some of the issues with this approach. First of all, we shouldn't expect a visitor to have to pore over the internal details of our complex structure of a building and then stumble their way around, when all they want to do is go to a room and sing a song.

There's also a security risk. What if they wander into a place they shouldn't be? Perhaps they see a whiteboard with "2024 Roadmap" and details like "Elon Musk to buy Disney in April", they take a picture, and then sell that info. Or, perhaps they wander into a dangerous part of the company and get hurt.

Finally, what if the appointment changed? Maybe Conference Room H has broken air conditioning, so Carol's party got moved to a different room. The team probably got notified via email or Teams, but you probably didn't, being an outsider. You might show up and there's no one there.

What's a better way?

There has to be a better way. Does anyone have an idea of what that might be?



The solution is actually pretty simple. Instead of forcing our visitor to navigate her own way around the complex structure that houses our business, we'll just simply lead her around to the places she needs to be.

The visitor no longer needs to know anything about how our company is laid out, and she doesn't need to go wandering around. All she needs to know is that if I follow that guy, he will take me to where I need to be.

The company is responsible for getting the visitor to the places she needs to go.

E.g., “Conference Room H”

In other words, the company is responsible for showing the visitor around. It will take her to different places in the company and then tell her “this is conference room H - do whatever you need to do here. Then I'll take you to conference room J, etc. Just follow me and I'll get you there.”

**The visitor only knows that
she'll be taken to the places
she needs to be so that she
can do her job there.**

The visitor is freed from the burden of reading maps and finding her way around the company. All she has to do is let the company take her to different rooms, and then she can do her business there.

That's the Visitor pattern in a nutshell.

It removes the burden of navigating a structure from visitors to the structure.

That's basically the Visitor design pattern. It moves the responsibility of navigating a structure away from the visitors to that structure.

Any questions so far?

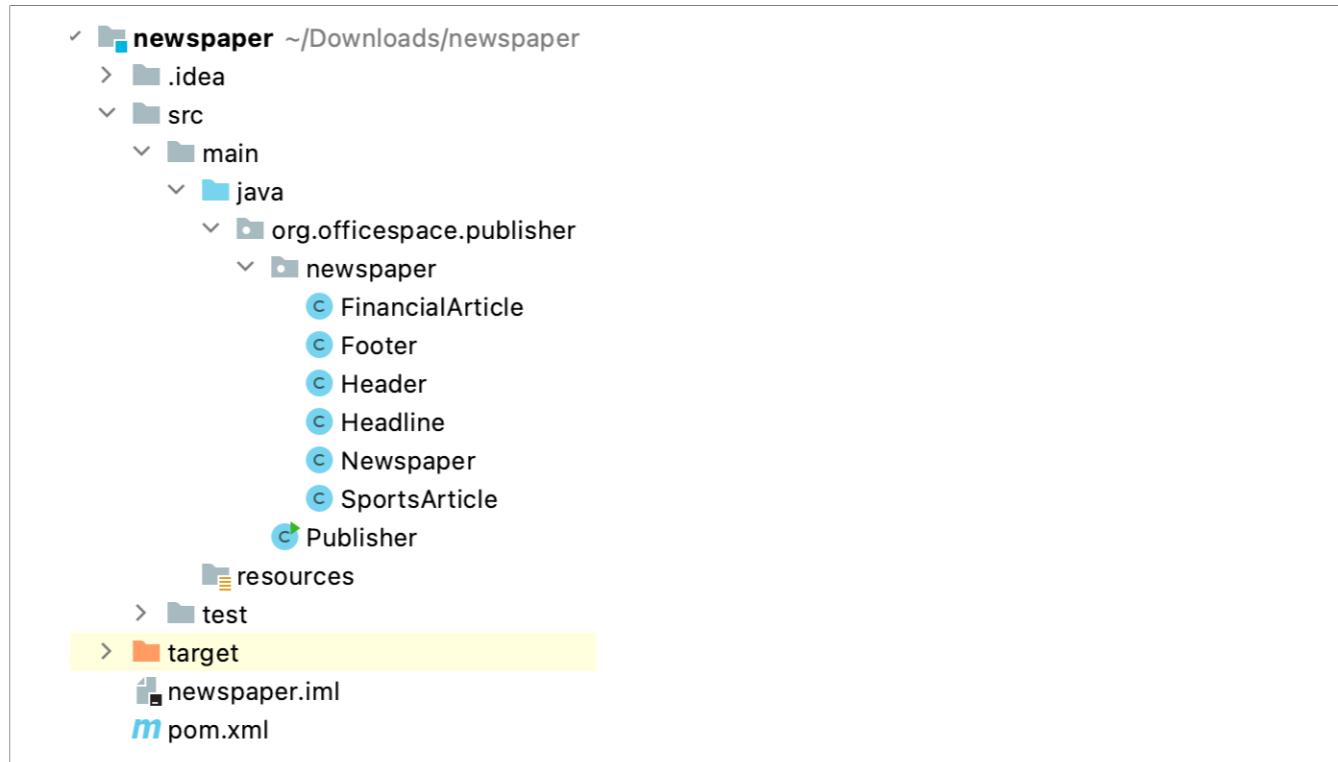


Fantastic! Okay, let's do that once more, but with code this time. (Anyone spot the cultural reference? *)

* title card from “Buffy the Vampire Slayer”: the “Once More, With Feeling” episode

**Let's imagine we're a
newspaper publisher
for the web...**

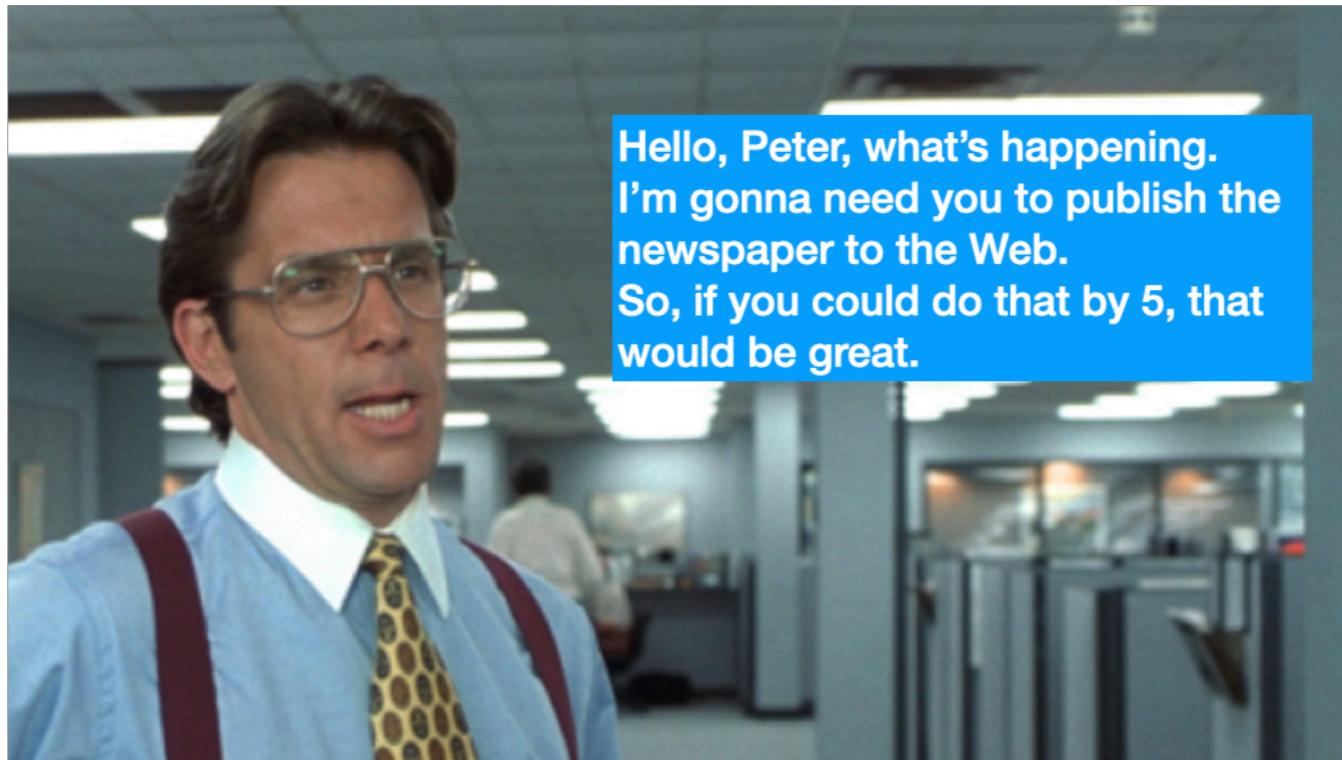
Let's give you another job! This time, we're going to be working for a company that publishes newspapers on the web.



Here's what that might look like in an IntelliJ Java project. We have a “Publisher” object that publishes a “Newspaper”. The “Newspaper” consists of several parts, a header and footer, some headlines, financial articles, and sports articles. Pretty straightforward, and good enough for our purposes in this presentation.

```
public static Newspaper createNewspaper() {
    return Newspaper.builder()
        .title("The Savannah Morning News")
        .header(Header.builder().title("The News For CodeMash 2023 - Thursday January 11").build())
        .headlines(Arrays.asList(
            Headline.builder().text("Design Patterns Are Still Cool").build(),
            Headline.builder().text("Whatever Happened To Ruby On Rails?").build()
        ))
        .sportsArticles(Arrays.asList(
            SportsArticle.builder().title("UGA Wins Another Natty").build(),
            SportsArticle.builder().title("TCU Had A Great Run").build()
        ))
        .financialArticles(Arrays.asList(
            FinancialArticle.builder().title("NASDAQ Down Again").build(),
            FinancialArticle.builder().title("401(k)s Not Looking Good").build()
        ))
        .footer(Footer.builder().copyright("An OfficeSpace Production (c) 2023").build())
        .build();
}
```

Here's how you might describe a `Newspaper` object in Java. I used the Lombok library to add builders to each of the objects, so the creation of a `Newspaper` can be accomplished in one line of code. And again, you can see how the `Newspaper` consists of a title, a header, some headlines, some sports articles, some financial articles, and a footer. Obviously, in real life, it would be much more complex, but again, this is sufficient for the purposes of this presentation.



Hello, Peter, what's happening.
I'm gonna need you to publish the
newspaper to the Web.
So, if you could do that by 5, that
would be great.

So, it's Monday morning, and you show up for work 15 minutes late and then zone out for about an hour, then your boss comes by. (Anyone do a good "Lumbergh" impression? [see if someone in the audience will read this slide]). He wants you to publish the newspaper to the web by 5.

```
public static String createHtmlVersion(final Newspaper newspaper) {
    StringBuilder html = new StringBuilder("<html><head></head><body>");
    html.append(toH1(newspaper.getTitle()));
    html.append(toH1(newspaper.getHeader().getTitle()));
    for (Headline headline : newspaper.getHeadlines()) {
        html.append(toH2(headline.getText()));
    }
    for (SportsArticle sportsArticle : newspaper.getSportsArticles()) {
        html.append(toH3(sportsArticle.getTitle()));
    }
    for (FinancialArticle financialArticle : newspaper.getFinancialArticles()) {
        html.append(toH3(financialArticle.getTitle()));
    }
    html.append(toH1(newspaper.getFooter().getCopyright()));
    html.append("</body></html>");
    return html.toString();
}

public static void main(final String[] args) {
    final String html = createHtmlVersion(createNewspaper());
    System.out.println(html);
}
```

Okay, that's not too bad. So, we fire up IntelliJ, and since we're software developers and do everything backwards, we start at the bottom and create a newspaper and then write a function to create an HTML version of the paper. You can see that we navigate around the newspaper, building up an HTML page, and then we print it out so that someone can deploy it to the web. I kept it simple here and just created the H1, H2, and H3 headers, but you can see how the full code would look. Not bad, and we go home at 5 and start up Netflix to see which of our favorite series they've canceled now.



The next day, you come in at least 15 minutes late and “zone out” for about an hour again. Then, like clockwork, your boss comes by and asks you to create a PDF version of the newspaper. So, since this sounds eerily familiar to what we did yesterday, what do we as software developers do?

CTRL-C
CTRL-V

Yep. Cut and paste.

```
public static String createPDFVersion(final Newspaper newspaper) {
    StringBuilder html = new StringBuilder(createPdfHeader());
    html.append(createPdfSection(newspaper.getTitle()));
    html.append(createPdfSection(newspaper.getHeader().getTitle()));
    for (Headline headline : newspaper.getHeadlines()) {
        html.append(createPdfSection(headline.getText()));
    }
    for (SportsArticle sportsArticle : newspaper.getSportsArticles()) {
        html.append(createPdfSection(sportsArticle.getTitle()));
    }
    for (FinancialArticle financialArticle : newspaper.getFinancialArticles()) {
        html.append(createPdfSection(financialArticle.getTitle()));
    }
    html.append(createPdfSection(newspaper.getFooter().getCopyright()));
    html.append(createPdfFooter());
    return html.toString();
}

public static void main(final String[] args) {
    final String pdf = createPDFVersion(createNewspaper());
    System.out.println(pdf);
}
```

Here's what it might look like. We basically copy the function that creates the HTML version and tweak it to generate PDF sections instead. In true "copy/paste" fashion, I neglected to rename the "html" variable, which is one of the potential pitfalls when copy/pasting code. You probably feel a little bad about writing what seems to be duplicate code, but it doesn't seem to be worth addressing at the moment, so you pack up, go home, and have a discussion through the wall with your next door neighbor.



Hello, Peter, what's happening.
Yeah, I'm gonna need you to create
an ePUB version for the Kindle.
So, if you could do that by 5, that
would be great.

Wednesday morning, your “Groundhog Day” life continues. You come in 15 minutes late, “zone out” for about an hour, and then your boss comes by like clockwork, coffee mug in hand, and asks you to create a Kindle version of the newspaper. Again, what do we do?

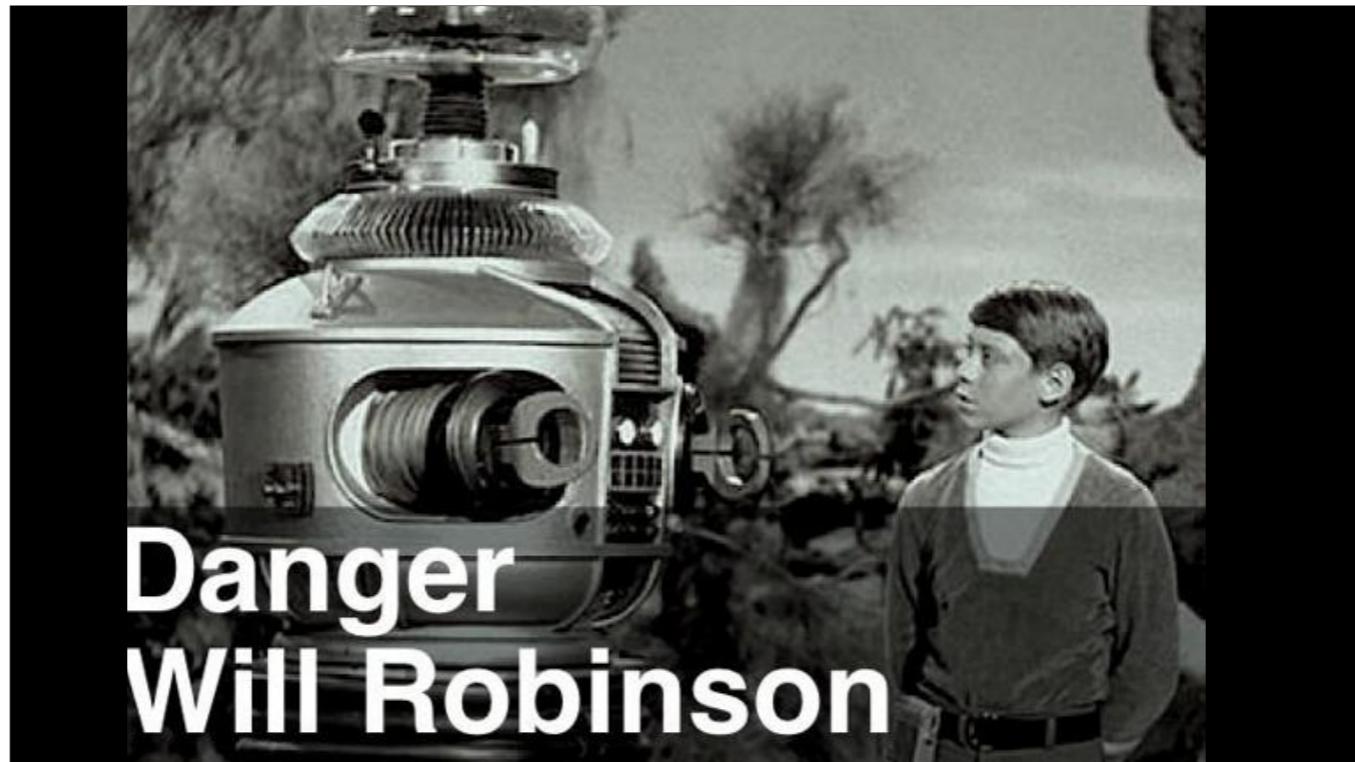
CTRL-C
CTRL-V

Say it with me: “Copy.” “Paste.”

```
public static String createEpubVersion(final Newspaper newspaper) {
    StringBuilder html = new StringBuilder(createEpubHeader());
    html.append(createEpubFragment(newspaper.getTitle()));
    html.append(createEpubFragment(newspaper.getHeader().getTitle()));
    for (Headline headline : newspaper.getHeadlines()) {
        html.append(createEpubFragment(headline.getText()));
    }
    for (SportsArticle sportsArticle : newspaper.getSportsArticles()) {
        html.append(createEpubFragment(sportsArticle.getTitle()));
    }
    for (FinancialArticle financialArticle : newspaper.getFinancialArticles()) {
        html.append(createEpubFragment(financialArticle.getTitle()));
    }
    html.append(createEpubFragment(newspaper.getFooter().getCopyright()));
    html.append(createEpubFooter());
    return html.toString();
}

public static void main(final String[] args) {
    final String epub = createEpubVersion(createNewspaper());
    System.out.println(epub);
}
```

Again, we copy our original function and replace the HTML generation lines with calls to create epub fragments.



**Danger
Will Robinson**

This is the third time we've copied/pasted/tweaked the same function.



**Your
“developer sense”
should be tingling**

Like Spider-Man, your “developer sense” should be tingling if you’re a good developer.



This is the moment that separates a “paycheck programmer” from a great developer. A paycheck programmer wouldn’t think twice about what we’ve written. But a good programmer would. So, since we’re good developers, let’s stop and think about our code and what’s wrong with it.

What's wrong with this code?

- **Duplication:** three functions that are all very similar
- We keep having to delve into **internal details** of the Newspaper object
- We have **navigation** code mixed with **business logic**
- It's **fragile**
 - What if the Newspaper internal structure changes?
 - What if we missed something (e.g., "Developer articles")?

The most obvious issue with our code is the duplication. Remember the DRY principle: “Don’t Repeat Yourself”. Well, we have. Also, even though we have a “Newspaper” object, we’re having to prowl around it to find things (like the sports articles) and then do something with them - that’s too much knowledge of the Newspaper class. Our functions are doing two things: they’re both navigating around the Newspaper class and then performing functions on the Newspaper. A good function should only do one thing, not two. And finally, it’s fragile - if the Newspaper data structure changes (e.g., from a List of articles to an array of articles, or a Hashmap of articles), then our code breaks. Also, something new might have been added that we don’t know about - maybe the newspaper started publishing developer articles while we were out on PTO.

What's the same? What's different?

Let's tackle the duplication problem. Any time you have duplication in your code, you need to ask yourself the same two questions. What's the same? and What's different?

```
public static String createPDFVersion(final Newspaper newspaper) {
    -StringBuilder html = new StringBuilder(createPdfHeader());
    -html.append(createPdfSection(newspaper.getTitle()));
    -html.append(createPdfSection(newspaper.getHeader().getTitle()));
    for (Headline headline : newspaper.getHeadlines()) {
        html.append(createPdfSection(headline.getText()));
    }
    for (SportsArticle sportsArticle : newspaper.getSportsArticles()) {
        html.append(createPdfSection(sportsArticle.getTitle()));
    }
    for (FinancialArticle financialArticle : newspaper.getFinancialArticles()) {
        html.append(createPdfSection(financialArticle.getTitle()));
    }
    html.append(createPdfSection(newspaper.getFooter().getCopyright()));
    html.append(createPdfFooter());
    return html.toString();
}
```

This prints the paper

This navigates the paper

Here's one of our functions. I've highlighted the parts that stayed the same from function to function, and the parts that changed. The yellow highlight parts were the same in all three functions. The blue part (is it blue? or purple? - I don't know: I'm a software developer, not a fashion consultant - this could be "mauve" or "fuchsia" for all I know). Anyway, the blue part is the code that differed from function to function.

This is similar to a visitor having to navigate an unfamiliar building



If we think back to our original example, this code is similar to that problem. We're having to navigate around a foreign data structure (the "Newspaper" object) when all we really want to do is to create some HTML, or a PDF, etc.

What if we separated those responsibilities?

So... what if we separated those responsibilities? What if we separated the NAVIGATION of the data structure from the USE of the data structure?

We'd like to pull the PDF generation code out of the navigation code

```
public static String createPDFVersion(final Newspaper newspaper) {  
    newspaper.getTitle()  
    newspaper.getHeader().getTitle()  
    for (Headline headline : newspaper.getHeadlines()) {  
        headline.getText()  
    }  
    for (SportsArticle sportsArticle : newspaper.getSportsArticles()) {  
        sportsArticle.getTitle()  
    }  
    for (FinancialArticle financialArticle : newspaper.getFinancialArticles()) {  
        financialArticle.getTitle()  
    }  
    newspaper.getFooter().getCopyright()  
}  
  
public static String createPDFVersion(final Newspaper newspaper) {  
    StringBuilder html = new StringBuilder(createPdfHeader());  
    html.append(createPdfSection());  
    html.append(createPdfSection());  
    html.append(createPdfSection());  
    html.append(createPdfSection());  
    html.append(createPdfSection());  
    html.append(createPdfSection());  
    html.append(createPdfFooter());  
    return html.toString();  
}
```

Time for a thought exercise. Remember the function with the yellow and, uh, “blue” highlighted parts? Let’s imagine that that code is a physical, tangible thing that we can hold in our hands. And let’s imagine that we grab the yellow code in one hand, and the blue code in our other hand, and we literally pull them apart. And when I do that, I imagine these terrible metallic shrieks and groans as I pull the code apart, but maybe that’s just me. When we finish, we have two functions like this.

The “intern” should do the navigation.
The “visitor” should just be led around to do her job.

```
public static String createPDFVersion(final Newspaper newspaper) {  
    newspaper.getTitle()  
    newspaper.getHeader().getTitle()  
    for (Headline headline : newspaper.getHeadlines()) {  
        headline.getText()  
    }  
    for (SportsArticle sportsArticle : newspaper.getSportsArticles()) {  
        sportsArticle.getTitle()  
    }  
    for (FinancialArticle financialArticle : newspaper.getFinancialArticles()  
        financialArticle.getTitle()  
    }  
    newspaper.getFooter().getCopyright()  
}
```





```
public static String createPDFVersion(final Newspaper newspaper) {  
    StringBuilder html = new StringBuilder(createPdfHeader());  
    html.append(createPdfSection( ));  
    html.append(createPdfSection( ));  
  
    html.append(createPdfSection( ));  
  
    html.append(createPdfSection( ));  
  
    html.append(createPdfSection( ));  
  
    html.append(createPdfSection( ));  
    html.append(createPdfFooter());  
    return html.toString();  
}
```

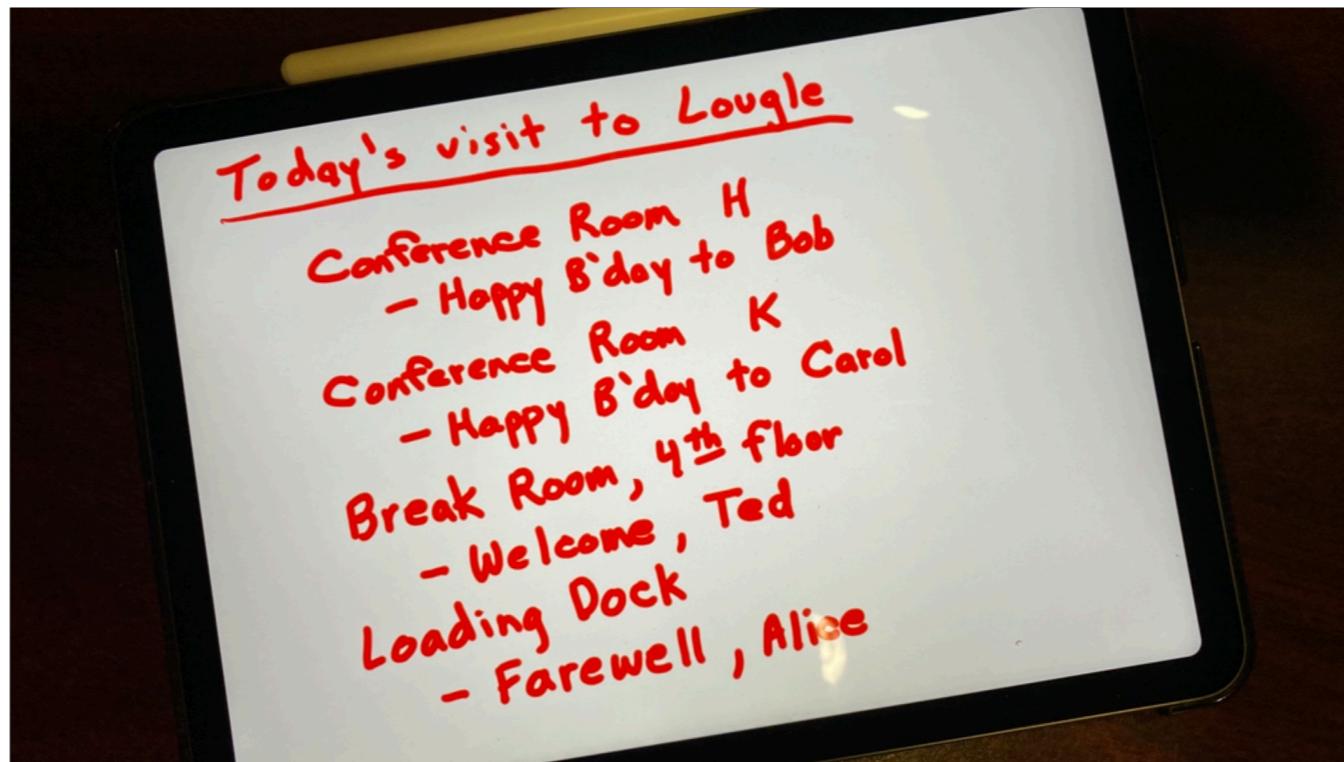


Now, the code on the left is navigation code. That's the part that the strangely familiar intern should be doing. The code on the right is code that prints the newspaper in a particular way. That's the part that the singer / visitor should be performing. Obviously, this ripped apart code doesn't work, and the two functions no longer are connected to each other, but you can kind of see where we're going with this. We want to only have the part that stays the same (the code on the left) written once, and the code on the right (the part that changes) to stay separate. How can we do this?

Let's start with the visitor



Let's start by addressing the part that changes from use to use (the visitor).



Remember our original agenda from the example? Where the visitor just wanted to make a visit to Lougle, then perform a task in four different locations within Lougle?



```
public class LougleVisitor {

    public void whenIAmInConferenceRoomH() {
        singHappyBirthdayTo("Bob");
    }

    public void whenIAmInConferenceRoomK() {
        singHappyBirthdayTo("Carol");
    }

    public void whenIAmInTheFourthFloorBreakRoom() {
        singWelcomeTo("Ted");
    }

    public void whenIAmOnTheLoadingDock() {
        singleGoodbyeTo("Alice");
    }
}
```

Here's what that might look like in Java code. We have a "LougleVisitor" class that represents the visitor's trip to Lougle. Inside the class, we have methods that are called when the visitor shows up at different rooms. In conference rooms H and K, we sing "happy birthday"; in the fourth floor break room, we sing a song welcoming Ted to the company; and on the loading dock, we sing "Hit the Road, Jack" to poor Alice (who is being "happy sized"). You'll notice that there is no navigation code in our Visitor class, and we're gone from an imperative style (where we had to navigate around) to a declarative style ("when I'm at conference room H, here's what I'll do").

```
public interface NewspaperVisitor {

    public default void begin() {}

    public default void doSomethingWithATitle(String title) {}

    public default void doSomethingWithAHeader(Header header) {}

    public default void doSomethingWithAHeadline(Headline headline) {}

    public default void doSomethingWithASportsArticle(SportsArticle sportsArticle) {}

    public default void doSomethingWithAFinancialArticle(FinancialArticle financialArticle) {}

    public default void doSomethingWithAFooter(Footer footer) {}

    public default void end() {}

}
```

Similarly, we can describe a “visit” to the `Newspaper` class. We provide a method to be called when we arrive at each part of the newspaper (e.g., “here’s a headline: what do you want to do with it?”). All of the methods are default methods with a no-op implementation - that means that if we don’t provide our own method when we implement this interface, then nothing will happen. Also, I added a `begin()` and an `end()` method - it’s very useful to be informed when a visit begins and ends, but it’s not necessary.

```
public class PDFVisitor implements NewspaperVisitor {

    StringBuilder pdf = new StringBuilder(createPdfHeader());

    @Override
    public void doSomethingWithAHeader(Header header) {
        pdf.append(createPdfSection(header.getTitle()));
    }

    @Override
    public void doSomethingWithATitle(String title) {
        pdf.append(createPdfSection(title));
    }

    @Override
    public void doSomethingWithAHeadline(Headline headline) {
        pdf.append(createPdfSection(headline.getText()));
    }

    @Override
    public void doSomethingWithASportsArticle(final SportsArticle sportsArticle) {
        pdf.append(createPdfSection(sportsArticle.getTitle()));
    }

    public String getPdf() {
        return pdf.toString();
    }
}
```



Here's how we can implement our generic "visit" to the `Newspaper` class to create a PDF version. We create a "PDFVisitor" class that implements the "NewspaperVisitor" interface, and then we create a PDF fragment as we're taken to each part of the newspaper. We also add a `getPdf()` method to retrieve the final PDF when we're done. Again, notice that we have declarative code instead of imperative code.

Now, let's put the navigation code in the Newspaper class



Okay, that takes care of the Visitor part of the code. Now let's see about taking that navigation code and making it reusable so we only have to write it once.

We want to move this navigation code into the Newspaper class

```
public static String createPDFVersion(final Newspaper newspaper) {  
    String result = newspaper.getTitle()  
        + newspaper.getHeader().getTitle()  
        + "  
    for (Headline headline : newspaper.getHeadlines()) {  
        result += headline.getText()  
    }  
    for (SportsArticle sportsArticle : newspaper.getSportsArticles()) {  
        result += sportsArticle.getTitle()  
    }  
    for (FinancialArticle financialArticle : newspaper.getFinancialArticles()) {  
        result += financialArticle.getTitle()  
    }  
    result += newspaper.getFooter().getCopyright()  
    return result;  
}
```



Remember the raw, bloody navigation code that we pulled out of our original function? The code that looked like this? Notice that it's all about the internal details of the `Newspaper` class - navigating around the `Newspaper`, accessing various parts of the `Newspaper` - so let's move it into the `Newspaper` class. Also, we need to hook this up with `Visitors` to the `Newspaper`.

```
public class Newspaper {  
  
    public void accept(NewspaperVisitor visitor) {  
        visitor.begin();  
        visitor.doSomethingWithATitle(title);  
        visitor.doSomethingWithAHeader(header);  
        for (Headline headline : headlines) {  
            visitor.doSomethingWithAHeadline(headline);  
        }  
        for (SportsArticle sportsArticle : sportsArticles) {  
            visitor.doSomethingWithASportsArticle(sportsArticle);  
        }  
        for (FinancialArticle financialArticle : financialArticles) {  
            visitor.doSomethingWithAFinancialArticle(financialArticle);  
        }  
        visitor.doSomethingWithAFooter(footer);  
        visitor.end();  
    }  
}
```



Here's how we can do that. We add an `accept()` method to the `Newspaper` class that accepts a `NewspaperVisitor`. Inside the method, we retain our navigation code, but now we call the methods on the `NewspaperVisitor` as we navigate to different parts of the data structure. I'll leave this code up for a moment so you can take a look at it. By the way, the "`accept(...)`" name is customary for the Visitor pattern, but not mandatory. You can name it whatever you'd like, but using "`accept(...)`" is an indication to other programmers that you're using the Visitor design pattern.

Finally, let's hook it all up

Okay. We've described a `NewspaperVisitor`, and we've made the navigation code reusable and we've moved it into our `Newspaper` class. Let's hook it all up!

```
public static void main(final String[] args) {  
  
    Newspaper newspaper = createNewspaper();  
  
    PDFVisitor pdfVisitor = new PDFVisitor();  
    newspaper.accept(pdfVisitor);  
  
    System.out.println(pdfVisitor.getPdf());  
}
```

And here we go! It's pretty straightforward. We create a `Newspaper`. Then we create an instance of a `PDFVisitor`, and we pass it to the `accept(...)` method of the `Newspaper`. When it returns, we retrieve the PDF from our `PDFVisitor`. Pretty straightforward. Remember at the beginning of the presentation when people thought this design pattern was complicated? It's really not.

What have we accomplished?

- The navigation code is now with the data structure itself, not with the users
- Users no longer need to know how to navigate the data structure
 - They just specify what they want to do for the items in it
- We can change the internal data structure without affecting users
- We can add new items to the data structure without breaking existing code
- We've separated the business logic from the data structure navigation

Let's stop for a moment and recap. [rare moment here when I actually just read a slide]

What if we wanted to create an HTML page?

```
public class HTMLVisitor implements NewspaperVisitor {

    private final StringBuilder html = new StringBuilder("<html><head></head><body>");

    @Override
    public void doSomethingWithAHeader(Header header) {
        html.append(toH1(header.getTitle()));
    }

    @Override
    public void doSomethingWithATitle(String title) {
        html.append(toH1(title));
    }

    @Override
    public void doSomethingWithAHeadline(Headline headline) {
        html.append(toH2(headline.getText()));
    }

    public String getHTML() {
        return html.append("</body></html>").toString();
    }

    private String toH1(final String text) {
        return "<h1>" + text + "</h1>";
    }
}
```

What if we wanted to create an HTML page instead of a PDF? Again, we just implement the `NewspaperVisitor` interface and then generate the relevant pieces of HTML as we're led around the structure. And again, this is declarative code, not imperative - there's no "navigation" code here, no looping around articles, etc. We just declare "when we get a headline, here's what we'll do". It's kind of hard to screw up code like that!

```
public static void main(final String[] args) {  
  
    Newspaper newspaper = createNewspaper();  
  
    HTMLVisitor htmlVisitor = new HTMLVisitor();  
    newspaper.accept(htmlVisitor);  
    System.out.println(htmlVisitor.getHTML());  
  
    PDFVisitor pdfVisitor = new PDFVisitor();  
    newspaper.accept(pdfVisitor);  
    System.out.println(pdfVisitor.getPdf());  
}
```

Using the HTMLVisitor is just like using the PDFVisitor. We create an instance, pass it to the accept(...) method of the Newspaper, then retrieve the HTML when we're done. You can see how we could also write visitors to create a Markdown version, or an RSS feed, etc. But we never have to write the navigation code again. We wrote the navigation code ONCE and added it to the Newspaper class. And the Newspaper is now free to tinker with the data structure: it can convert a List of articles to an array of articles, or a B-tree of articles, or a HashMap of articles... and our code NEVER CHANGES. We just trust the Newspaper to get to an article however it wants to do it, and then it will tell us "here's an article: do something with it."

We can do some interesting things...

```
public class ArticleCounter implements NewspaperVisitor {

    private int sportsArticleCount;

    public void doSomethingWithASportsArticle(SportsArticle sportsArticle) {
        ++sportsArticleCount;
    }

    public int getSportsArticleCount() {
        return sportsArticleCount;
    }
}
```

We can also do some cool things. What if we wanted to count the number of sports articles? We can create a visitor (“ArticleCounter”) and then just increment a counter each time we visit a sports article. And remember, we provided default methods for everything in the `NewspaperVisitor` interface, so we only have to override the method we’re interested in (“`doSomethingWithASportsArticle(...)`”).

This could be the beginning of a statistics class

```
Newspaper newspaper = createNewspaper();

ArticleCounter articleCounter = new ArticleCounter();
newspaper.accept(articleCounter);
System.out.println("There are " + articleCounter.getSportsArticleCount() + " sports articles.");
```

Using it is just as easy as creating the HTML or PDF version. You can see how this could be the beginning of a Newspaper statistics class.

We could use this for filtering

```
public class OnlyShowGeorgiaSportsArticles implements NewspaperVisitor {  
  
    @Override  
    public void doSomethingWithASportsArticle(SportsArticle sportsArticle) {  
        if (sportsArticle.getTitle().contains("Georgia")) {  
            showArticleOnFrontPage(sportsArticle);  
        } else {  
            /* ignore it */  
        }  
    }  
}
```

We could also use it for filtering. My Georgia Bulldogs just won a second national championship in college football, so maybe I want to publish sports articles containing the word “Georgia” to the front page and then ignore everything else. Here’s how I could accomplish that.

That's the Visitor pattern*

So, yeah. That's pretty much the Visitor pattern. With one slight difference...

The “Gang of Four” pattern is slightly different.
There, each object in the hierarchy implements the accept(...) method.

The description of the Visitor pattern in the “Gang of Four” book is slightly different. In my implementation, I just added an accept(...) method to the top level Visitor class. In the “Gang of Four” implementation, an accept(...) method is added TO EACH OBJECT IN THE DATA STRUCTURE. Which means that the Header and Footer have accept(...) methods, the SportsArticle and FinancialArticle objects have accept(...) methods, etc.

```
@Builder  
@Getter  
public class FinancialArticle {  
  
    private final String title;  
  
    public void accept(NewspaperVisitor newspaperVisitor) {  
        newspaperVisitor.doSomethingWithAFinancialArticle(this);  
    }  
}  
  
@Builder  
@Getter  
public class SportsArticle {  
  
    private final String title;  
  
    public void accept(NewspaperVisitor newspaperVisitor) {  
        newspaperVisitor.doSomethingWithASportsArticle(this);  
    }  
}
```

It's not a major change. Here's what it looks like (btw, the @Builder and @Getter annotations are Lombok annotations that generate a Builder pattern and getter methods for the classes).

```

public class Newspaper {
    public void accept(NewspaperVisitor visitor) {
        visitor.begin();
        visitor.doSomethingWithATitle(title);
        visitor.doSomethingWithAHeader(header);
        for (Headline headline : headlines) {
            visitor.doSomethingWithAHeadline(headline);
        }
        for (SportsArticle sportsArticle : sportsArticles) {
            visitor.doSomethingWithASportsArticle(sportsArticle);
        }
        for (FinancialArticle financialArticle : financialArticles) {
            visitor.doSomethingWithAFinancialArticle(financialArticle);
        }
        visitor.doSomethingWithAFooter(footer);
        visitor.end();
    }
}

public class Newspaper {
    public void accept(NewspaperVisitor visitor) {
        visitor.begin();
        visitor.doSomethingWithATitle(title);
        header.accept(visitor);
        for (Headline headline : headlines) {
            headline.accept(visitor);
        }
        for (SportsArticle sportsArticle : sportsArticles) {
            sportsArticle.accept(visitor);
        }
        for (FinancialArticle financialArticle : financialArticles) {
            financialArticle.accept(visitor);
        }
        footer.accept(visitor);
        visitor.end();
    }
}

```

Here's how we'd change my `accept(...)` method (on the left) to match the "Gang of Four" method (on the right). The only difference is that we now call the `accept(...)` method for each of the parts of the `Newspaper` data structure (and then they will call back to the `NewspaperVisitor` object) instead of calling the `NewspaperVisitor` methods directly.

TypeScript implementations

“Hey, Kelly - you promised there would be TypeScript implementations in this presentation.”

Yes, I did. So let’s take a look at implementing the Visitor design pattern in TypeScript.

```
class Header {
    private title: string;
}

class Headline {
    private title: string;
}

class SportsArticle {
    private title: string;
}

class FinancialArticle {
    private title: string;
}

class Footer {
    private title: string;
}

class Newspaper {
    private title: string;
    private header: Header;
    private headlines: Headline[];
    private sportsArticles: SportsArticle[];
    private financialArticles: FinancialArticle[];
    private footer: Footer;
}
```

Here's what a TypeScript implementation of our Newspaper data structure might look like. I'll leave it up for a moment so you can look at it.

Let's use ChatGPT3!

Now - rather than boring you by writing the code again, let's do something fun and different - let's use ChatGPT3! It's 2023 - I can't do a presentation without at least mentioning ChatGPT3.



show me how to add the visitor design pattern to this TypeScript code

```
class Header {  
    private title: string;  
}  
  
class Headline {  
    private title: string;  
}  
  
class SportsArticle {  
    private title: string;  
}  
  
class FinancialArticle {  
    private title: string;  
}  
  
class Footer {  
    private title: string;  
}  
  
class Newspaper {  
    private title: string;  
    private header: Header;  
    private headlines: Headline[];  
    private sportsArticles: SportsArticle[];  
    private financialArticles: FinancialArticle[];  
    private footer: Footer;  
}
```

Here's the ChatGPT3 prompt I used. I just said "Show me how to add the visitor design pattern to this TypeScript code" and then pasted in my TypeScript data structure. Then I submitted it. A few seconds later, guess what I got back?

Here is an example of how you might add the Visitor pattern to the given TypeScript code:

```
interface Visitor {
    visitHeader(header: Header): void;
    visitHeadline(headline: Headline): void;
    visitSportsArticle(sportsArticle: SportsArticle): void;
    visitFinancialArticle(financialArticle: FinancialArticle): void;
    visitFooter(footer: Footer): void;
}

class Header {
    private title: string;

    accept(visitor: Visitor): void {
        visitor.visitHeader(this);
    }
}

class Headline {
    private title: string;

    accept(visitor: Visitor): void {
        visitor.visitHeadline(this);
    }
}

class SportsArticle {
    private title: string;

    accept(visitor: Visitor): void {

```

```
class Footer {
    private title: string;

    accept(visitor: Visitor): void {
        visitor.visitFooter(this);
    }
}

class Newspaper {
    private title: string;
    private header: Header;
    private headlines: Headline[];
    private sportsArticles: SportsArticle[];
    private financialArticles: FinancialArticle[];
    private footer: Footer;

    accept(visitor: Visitor): void {
        this.header.accept(visitor);
        this.headlines.forEach(headline => headline.accept(visitor));
        this.sportsArticles.forEach(sportsArticle =>
            sportsArticle.accept(visitor));
        this.financialArticles.forEach(financialArticle =>
            financialArticle.accept(visitor));
        this.footer.accept(visitor);
    }
}
```

In this example, the `Header`, `Headline`, `SportsArticle`, `FinancialArticle`, and `Footer` classes have been given an `accept()` method that takes a `Visitor` as an argument. The `Newspaper` class also has an `accept()` method that calls the `accept()` method of each of its properties, including the arrays of `Headline`, `SportsArticle`, and `FinancialArticle`.

Wow - it did a pretty good job! You can see that it created a Visitor (which looks remarkably like the `NewspaperVisitor` in our Java example). It added `accept(...)` methods to each of the parts of our data structure, like the canonical “Gang of Four” implementation. And it created an `accept(...)` method in our `Newspaper` class that does the top-level navigation of the structure. Amazing, really. But (and there’s always a “but”), it’s not perfect. If you look at the `Newspaper` class, you’ll see that it navigates to all of the instance fields that are objects, but it skipped over the instance field that’s a primitive (the “`title`”). So, instead of blindly copy/pasting this code, we would need to add a `visitTitle(...)` method to the `Visitor` class, and then call it with the `title` in the `Newspaper`’s `accept(...)` method. Fairly trivial, but again, a good caveat not to blindly trust anything that ChatGPT3 spits out.

You can create a visitor class that implements the Visitor interface, and then call the accept method of the newspaper object, passing the visitor as a parameter, in this way the visitor will visit all the components of the newspaper.

 Copy code

```
class PrintVisitor implements Visitor {
    visitHeader(header: Header): void {
        console.log(`Header: ${header.title}`);
    }
    visitHeadline(headline: Headline): void {
        console.log(`Headline: ${headline.title}`);
    }
    visitSportsArticle(sportsArticle: SportsArticle): void {
        console.log(`Sports Article: ${sportsArticle.title}`);
    }
    visitFinancialArticle(financialArticle: FinancialArticle): void {
        console.log(`Financial Article: ${financialArticle.title}`);
    }
    visitFooter(footer: Footer):
```

ChatGPT3 also created an example Visitor implementation (PrintVisitor) that logs the items in the Newspaper to the console. Nice.



That's pretty much all I wanted to talk about today. Again, my name is Kelly Morrison, and I'll stick around for any questions you might have.

Daugherty

BUSINESS SOLUTIONS

Work: kelly.morrison@daugherty.com

Home: kellyivymorrison@gmail.com

Kelly Morrison
Manager / Application Architect at Daugherty
Business Solutions



[LinkedIn](#)



Also, let's keep in touch. Here are my work and personal emails, and a QR code that takes you to my LinkedIn profile. And yes, my middle name is "Ivy" - I tell people that it's "Ivy as in League, not as in poison". :) My company (Daugherty Business Solutions) has a booth here if you're interested in joining us. And I'll leave you a parting shot of my dog, "Daisy Mae", who personally reviewed all of the code that I wrote for this presentation. Thank you!