Extracted from:

Creating Software with Modern Diagramming Techniques

Build Better Software with Mermaid

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Pragmatic exPress

Creating Software with Modern Diagramming Techniques

Build Better Software with Mermaid

Ashley Peacock

Edited by Michael Swaine

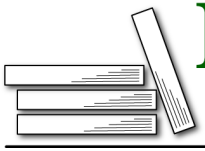
Creating Software with Modern Diagramming Techniques

Build Better Software with Mermaid

Ashley Peacock

The Pragmatic Bookshelf

Raleigh, North Carolina



For our complete catalog of hands-on, practical, and Pragmatic content for software developers, please visit *https://pragprog.com*.

The team that produced this book includes:

CEO: Dave Rankin
COO: Janet Furlow
Managing Editor: Tammy Coron
Development Editor: Michael Swaine
Copy Editor: L. Sakhi MacMillan
Layout: Gilson Graphics
Founders: Andy Hunt and Dave Thomas

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

ISBN-13: 978-1-68050-983-0
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—February 2023

In the following chapters, we're going to go through the life cycle of creating an application from scratch. We won't be writing any application code, but we will be documenting the important steps with diagrams. I've written the chapters roughly in the order I would go about each step in my professional career, but you may find a slightly different order works for you.

To start off, we're going to create a domain model. Domain modeling is the primary way of determining the important aspects of a business. It's usually created collaboratively by engineering, product, and business stakeholders to ensure all major parts of the business are aligned on what the domain model looks like.

That makes it a good candidate for diagramming. By documenting your domain models with a diagram, that domain model is going to come to life and is more likely to be practiced. Furthermore, due to the ease of Mermaid, it's possible to create a draft of the domain model in real time when discussing with colleagues in a meeting.

I recommend reading *Domain-Driven Design: Tackling Complexity in the Heart of Software [Eva03]* if you're keen to learn more about DDD.

Once you've got to grips with domain modeling, it's going to make your life so much easier. There will be moments during domain modeling where suddenly it all comes together, everyone is on the same page in terms of their understanding of the landscape, and everyone is speaking the same language.

The most powerful use of DDD I've experienced was working for an insurance company. The team I was working in had been tasked with creating a way to determine how exactly our products were sold and to whom—sounds easy, right? Unfortunately, the data was all over the place, and customers could use many avenues to make a purchase, with varying data available in each avenue. None of us had any idea how to represent these conceptions in our code, so we spent several days domain modeling. We tried out different ideas and approaches, and in the end we landed on a culmination of a few different ideas and were able to easily translate the domain model into code.

I find the biggest power that comes from domain modeling is the collaboration, how it brings everyone on the journey and ultimately to the same destination. It becomes easy to talk about what we are working on, as we're all speaking the same language, and the code flows easily because we have a clear idea of how we should represent these business requirements using our domain model. By the end of the project, even nontechnical stakeholders were using the same terminology that we had come up with while domain modeling.

The final selling point I'll make for DDD is its ability to allow your domain and code to evolve over time. Because we're documenting our domain model, we can reference it at any point. It's very common for new requirements to come along later, at which point we can refer back to the domain model, see if it fits the new requirements, and if not, it can evolve and adjust as necessary. The core of the domain we modeled at the insurance company is still intact today but has since evolved to add new entities and use cases.

Within UML, one type of diagram available to us is called a class diagram. It can be used to model classes, but it can also be used to model domains, which makes a lot of sense when you consider your domain model is implemented in your codebase with classes. The real power of the diagram is realized when we start to model relationships between entities, which we can easily do with a UML class diagram.

Determine the Important Entities

Firstly, when creating your domain model, think of all the important entities within your business. For those unfamiliar with the term, an entity represents a core concept within the business. Entities are typically the phrases that are most used in the codebase and in meetings. A book publisher, for example, would likely have entities such as book, chapter, and author. We won't be going to this level of detail, but in your codebase they would contain entity data and business logic. Continuing the publishing example, a book would have a title for data and perhaps some business logic that calculates the word count.

The fictional company I'll be using through these chapters is called Streamy, which is trying to make a name for itself in the video streaming industry.

I would say for a video streaming company, its most important entity is likely to be `Title`—representing the actual videos Streamy offers their customers.

Once you've thought of an entity, what related entities might you have?

Typically, each `Title` will belong to a `Genre`, and each `Genre` will have a list of `Titles` associated with it. We've now identified two entities and how those two entities are related to one another, so we can start to form our domain model next.

Domain-Driven Design

Alongside domain modeling sits domain-driven design (DDD), which is a methodology not only for determining the domain model but keeping that domain model alive in the codebases you work on by ensuring the entities in your domain model are represented in your codebase.

If you're not familiar with either construct, I highly recommend reading about them after reading this book, but even without knowing DDD in depth, you can have a go at creating a domain model later in the chapter.

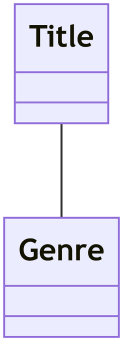
Document Our First Relationship

Now that we understand what domain-driven design is and what it's used for, we can begin to create a domain model for Streamy.

We're going to use Mermaid to create our domain model, and adding these two entities is super-simple:

```
classDiagram
    Title -- Genre
```

That's it—our first two entities are defined! If we generate the diagram, it looks like this:



In Mermaid, the markup for any diagram (with the exception of adjusting configuration, which we will cover later) is the type of diagram you wish to create. In our case, that's a `classDiagram`, which informs Mermaid how to format and interpret the following lines.

When creating a domain model in UML, each line after the initial line documents a relationship between two entities. In the case of `Title` and `Genre`, where each entity is going to hold a reference to the other, that type of relationship is known as an *association*. In Mermaid, that's where the second line comes in:

```
Title -- Genre
```

To define any relationship, we write down the two entities separated by a set of symbols that define the type of relationship. An association is defined using two hyphens. Let's look at associations in more detail.

Define Associations

Association is the first type of relationship we'll cover, and it's the loosest type of relationship available within a UML class diagram. Usually, for a relationship to be classed as an association, the entities must be able to exist independently of one another and likely have their own life cycles. Furthermore, there is generally no "owner" of the relationship for associations, they're simply linked. In that way, you can think of their relationship as "using" one another rather than one owning the other.

In our case of Title and Genre, we could have a page on our application that just shows Genres without any Titles. However, we probably want to be able to list the Titles within a Genre, and display a specific Title's Genre too, so they need to have a link of some sort, as all relationships must be documented in a domain model.

Associations are very loose ways to link two entities, but some relationships form a closer bond between two entities. Let's look at one next.

Define Composite Relationships

Once you have your first two entities defined, think of other entities that relate to either of them. We're going to focus on Title, as that's our main entity. I can think of two related entities:

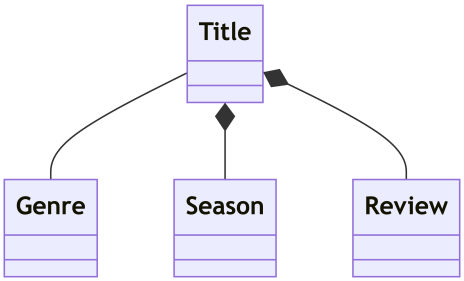
- Season
- Review

These two entities don't suit an association, though. Similarly to associations, we have to think about how each of these behave in relation to Title. They don't make sense on their own like an association, and without Title, they probably make no sense at all in our domain. After all, if we deleted Title, it would be impossible to have either of these exist. For example, you can't have a Season for a nonexistent Title, and the same is true for Review.

This type of relationship is called *composition*, and we can document it as follows in Mermaid, using an asterisk and two hyphens:

```
classDiagram
    Title -- Genre
    Title *-- Season
    Title *-- Review
```

And if we generate the diagram, it looks like so:



Notice the difference between the different classes. Title and Genre are linked via a solid line with no arrows, whereas Title and Season have a solid diamond on the line connecting them, which signifies the relationship is composition.

The side the diamond is on indicates the class holding the reference, but you can think of it similarly to a parent and child relationship, where the diamond signifies the parent, as the child cannot exist without the parent. Unlike associations, the parent is the owner of the relationship.

Relationship Direction

While Mermaid will allow you to write the relationships in either direction (for example, '*–' and '–*'), I recommend always putting the parent on the left for easier readability and less cognitive load of working out the direction of the relationship, as you can just read from left to right.

To complete the composite relationships we need for our domain model, a Season isn't much use without its counterpart Episode. Much like our other composite relationships, an Episode *probably* doesn't make much sense without belonging to a Season. Once adding our final composite relationship, our Mermaid code looks like so:

```
classDiagram
    Title -- Genre
    Title *-- Season
    Title *-- Review

    Season *-- Episode
```

While the layout is totally up to you and personal preference, I prefer to group each entity's relationships together and separate the groups with an empty line to aid readability.

DDD Is Opinionated

> Domain-driven design is very opinionated, so keep in mind you might not model my examples like this if you were creating them. The model itself isn't the important part, at least in the context of this book—the syntax and the diagrams themselves are.

We now know two relationship types:

- *Associations*, which are two entities that are loosely related and can exist independent of one another.
- *Compositions*, which indicate two entities are tightly related and cannot exist independently of one another.

One more, though, sits between those two in terms of how closely related two entities are. We'll cover that in the next section.

Define Aggregate Relationships

Our domain model is starting to take form, but it's still missing a few key entities. After all, doesn't everyone like to know the Actors that appear in a Title?

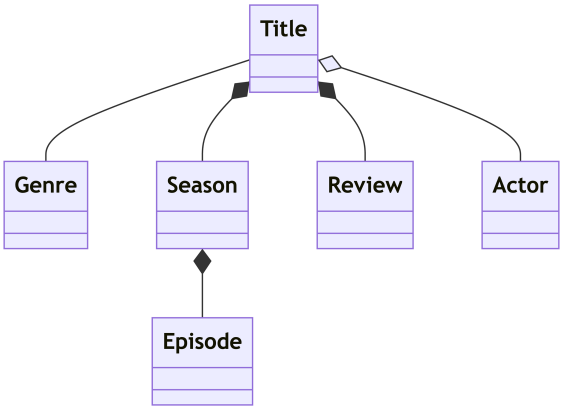
But this entity doesn't suit a composite relationship. While knowing the Actors is definitely handy, Title could exist without Actors, and similarly an Actor can exist independently of a Title (or belong to many Titles, so if one was deleted, they'd perhaps exist on another, so would remain in place).

Let's update our Mermaid code to add Actor:

```
classDiagram
    Title -- Genre
    Title *-- Season
    Title *-- Review
    Title o-- Actor

    Season *-- Episode
```

The syntax for *aggregate* relationships is `o--` (the letter `o` followed by two hyphens), and once generated this is how they are rendered:



An aggregate relationship is also displayed as a diamond, but instead of being a solid diamond, it's empty. In an aggregate relationship, there's still an owner—the parent. However, the bond between them isn't as strong as a composite relationship, and if the parent were to be deleted, the child can still exist.

As mentioned at the start of the chapter, choices of how to model a domain can vary greatly between companies or colleagues. In another domain model, perhaps you group `Actors` into a `Cast` for example. That would change the type of relationship, perhaps, to a composite relationship, as a `Cast` created for a `Title` probably wouldn't remain if the `Title` were deleted.

Now that we know the main three types of relationship we can use, let's consider when to use each one.