Chapter 4. Modeling

All models are wrong, but some are useful.

George E.P. Box, British statistician (attributed)

By now, it should be clear that communication is a central focus of your work as a software engineer, especially communication between you and other developers. While the computer cares only about syntactically correct code, communicating with other humans takes much more. Your code should be well-documented and organized so it can be understood by other people (see Chapter 3 for more on writing code), but sometimes you'll want more. Throughout a project, you will use software models or box and line diagrams to express your technical intent.

Much like good code, good software models are clear and easy for your stakeholders to understand. If your models aren't clear, those consuming your models won't understand your technical intent. There is no shortage of consumers for your models: users, testers, other developers, security, the people writing the checks, project managers, and architects. Yourself. Sometimes, the only consumer of a diagram will be you. That said, you can't expect to draw some pictures and expect everyone to understand them. A diagram that's perfect for a developer might not work so well for the vice president of engineering. And vice versa. Your challenge is to know what diagram to create and when to create it.

What Is Software Modeling and Why Do We Do It?

Software is a relatively young industry, and as such, has borrowed concepts and approaches from more mature disciplines.[1] There have also been various "waves" of modeling approaches, from the Unified Modeling Language (UML) to the C4 model. The construction industry creates a full set of blueprints before breaking ground on a new project; shouldn't software projects do the same? Maybe.

Software modeling is the process of creating abstract representations of a software system to better understand, analyze, and communicate its structure, behavior, and functionality. These models guide developers, designers, and stakeholders through the system's design and development process. Good software models reflect the real-world problem domain, providing insight throughout the development process.

However, it is important to understand the fundamental differences between writing software and building a house. Refactoring the physical world is difficult and expensive: a builder cannot afford to test the viability of a load-bearing wall in production, as it were. Blueprints allow the designers to ensure that the house meets local building codes and has all the appropriate routing for water, electricity, and heating/cooling. Before construction starts, the blueprints communicate what is being built, allowing the owner to agree that everything is as they expect. When construction has finished, the blueprints can be used to confirm that everything was built to specification. Imagine the chaos on a building site if all the various contractors were left to improvise.

Software, however, can be refactored.[2] Code can be changed in a few hours for a few hundred dollars. The cost of "refactoring" the physical world is considerably higher; arguably, houses would be built differently if the cost of nearly every minor change was a few extra hours and a few hundred dollars. Blueprints wouldn't be nearly as integral to the building process. Because software is more malleable, diagramming may not be as critical to project success.

Diagrams don't compile; they don't result in executable code that contributes to the completion of your project. In fact, you can make a pretty strong argument that for developers your code is the ultimate design artifact. Jack Reeves, author of the influential Code as Design papers, argues that programming is fundamentally a design activity and that the purest expression of that design is the code itself.

If diagrams don't compile, why are they useful to you? Diagrams can provide context. They can be used to understand and manage the complexity of a system and decompose the problem. You can use them to predict quality attributes, otherwise known as the nonfunctional requirements or the abilities that you learned about in Chapter 9. Diagrams can help you design for certain quality attributes.

Some organizations require you to create various diagrams as part of your SDLC. These requirements can be a blessing and a curse.

Diagrams can help you plan your system design. If drawing a picture helps you wrap your head around a design, go ahead and draw the diagram. An hour or two sketching out a solution could save you days of development time.

Diagrams can also help during a software archaeology expedition, such as when you are learning about a new system or first being exposed to it. Of course, that assumes those diagrams are accurate. Diagrams can be useful when onboarding new engineers. Again, assuming they're accurate. They can also be useful when transferring knowledge to a new member of the team, once again assuming they're accurate. Diagrams can be critical when debugging a system, as knowing the boundaries of the system and its core responsibilities can help you design test cases or determine how to approach debugging.

As you might have figured out by now, a lot depends on whether your diagrams are accurate and up-to-date. If a diagram isn't clear or representative of the current state of the code, its value is greatly diminished.[3] That leads to a rather interesting question: how permanent are diagrams? One could argue that diagrams should have an expiration date and that it is perfectly acceptable to throw a diagram away once it no longer proves useful. They can be as ephemeral as a sketch on a whiteboard. They could also be formal and made with a modeling tool.

Tip

If you find yourself drawing the same diagram again and again, that's a pretty good indication that you should formalize it in some way, shape, or form. Take the time to create it with a tool and store it centrally for your project; if it's helpful to you, it'll be useful for a teammate too. For larger diagrams, leverage a plotter printer; a physical copy hanging on the wall can be useful for the entire team. In some cases, printing a diagram on cardstock can make it feel more real and valuable to people.

Of course, you could also just take a picture of the diagram and add it to your project documentation if you prefer. Don't let a tool slow you down.

There is no shortage of modeling tools at your disposal. They range from simple diagramming tools to high-end, full-featured enterprise modeling tools. Massive projects with large teams spread across the globe can find significant value in standardizing on a given enterprise tool. Some tools are for drafting models by hand, while others can generate models from code.[4] You'll explore tools later in this chapter, but fancy tools don't mean that you'll have better diagrams that'll suit your needs.

Besides tools, there are many diagram types to choose from as well.

Which Diagrams Do You Need?

When it comes to software modeling diagrams, you have a plethora to choose from. The challenge is to know which diagram to create and when to create them throughout the software development process. In other words, which diagrams do you need? Which ones are most important at this moment during a project? Of course, the only answer that we can give you is, "It depends." What are you trying to do? How complicated is the problem? How risky is the application? Have you ever built an app like this before, or is this novel? What is the project budget? Is this project critical to the business?

Diagrams can be formal or informal. Formal models use technical notation, such as UML (Figure 4-1). You must understand the audience for your model. The more formal the model, the more technical the audience has to be to consume it. This typically means a smaller subset of people will understand your intent. For example, do you understand Z notation?[5] Even if you understand it perfectly, how widely known is it within your organization?

The less formal the method, the less technical your audience will need to be to understand what you are trying to communicate. This allows for a larger audience that can consume your diagram.

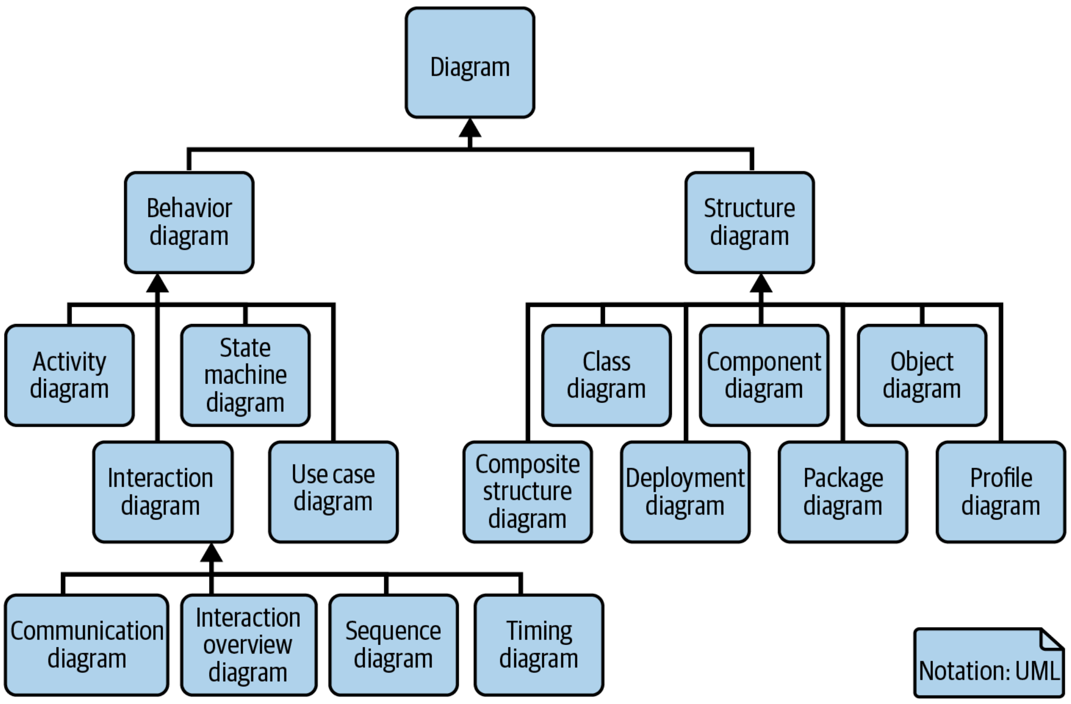

Figure 4-1. Standard UML diagrams, adapted from Wikimedia Commons

Unified Modeling Language

UML was developed in the mid-1990s as a way of harmonizing various other notation systems developed in the 1980s and early 1990s. The Object Management Group adopted it as a standard in 1997, and it was published as the ISO/IEC 19501 standard in 2005. That said, most software engineers prefer informal diagrams (which often use UML elements) instead. Why?

Do you know the difference between a filled diamond and an empty diamond in a UML diagram?[6] You may completely understand the nuance of this particular feature. However, your audience may not. Most software engineers choose to use informal notation over UML in the interest of reaching a broader audience. Do not shun what works. Simple is almost always better.

Context Diagrams

Context diagrams define the boundary between systems or parts of a system. They show the environment as well as how entities interact. These are logical data entities that may often include data at least in terms of volume and frequency. Context diagrams are very high level.

Context diagrams are frequently used by architects, leadership, project managers, and product owners. They are often used early in a project to define boundaries and get people on the same page about what is and is not included within a given application. They provide a useful overview of what we mean by a given system, and they quickly show you the edges of the map.

Think of context diagrams as the most zoomed-out view of a system. For example, Figure 4-2 is a context diagram for a system that organizes and manages all the data from a self-driving car. The car sends back data that can be analyzed by data scientists and also pushes notifications to the owner of the vehicle.

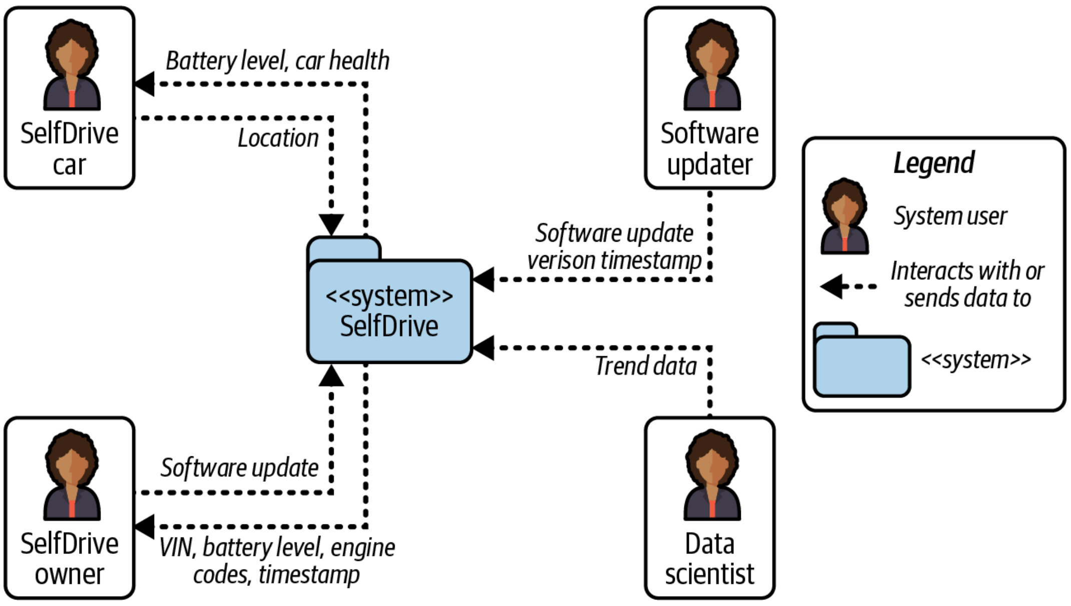

Figure 4-2. Sample context diagram

Component Diagrams

Component diagrams show the principal elements of a system at runtime. They show how the system works together, illustrating structure and behavior. They show information flows and interfaces. Component diagrams are often used by developers, architects, production support, and DevOps engineers.

Component diagrams are used throughout the project. Originally, they were designed to define the expected interactions, but they are also useful for telling the story of the project to a more technical audience. They can be very useful in knowledge transfer and onboarding.

For example, in Figure 4-3, you have a system that acts as a directory of available APIs. There are two separate views, one specific to administrators of the system and another for consumers and creators of the APIs. There is also a user directory for authentication and authorization, and the information is stored in a database.

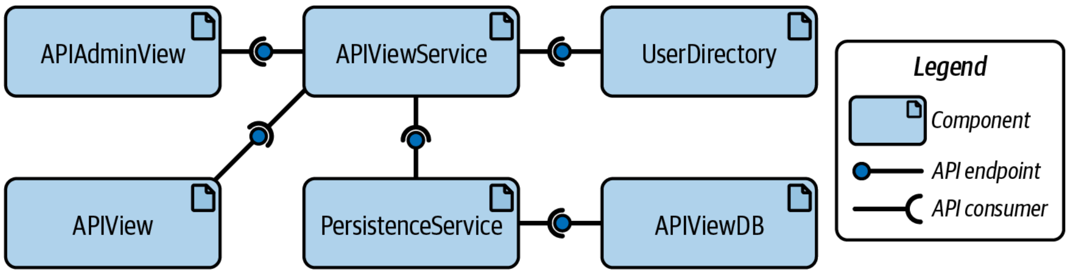

Figure 4-3. Sample component diagram

Class Diagrams

As the name implies, *class diagrams* show how your classes relate to one another. They show inheritance relationships as well as composition relationships and can include cardinality.[7] They show logical entities and can include methods where helpful.

Class diagrams are often extracted from existing code as needed because once created, they will quickly become out-of-date with the code. They can be overwhelming on large systems and may be broken into logical or domain boundaries to make them more consumable.

The audience for class diagrams is very technical in nature—usually other engineers, architects, dev, ops, etc. Class diagrams are often created early in a project to help people understand the overall picture and are typically refined throughout the project. They can be very helpful for both new and existing developers to understand the full scope of the system. For example, the class diagram in Figure 4-4 shows that `Person` and `NamedEntity` are subclasses of `BaseEntity`.

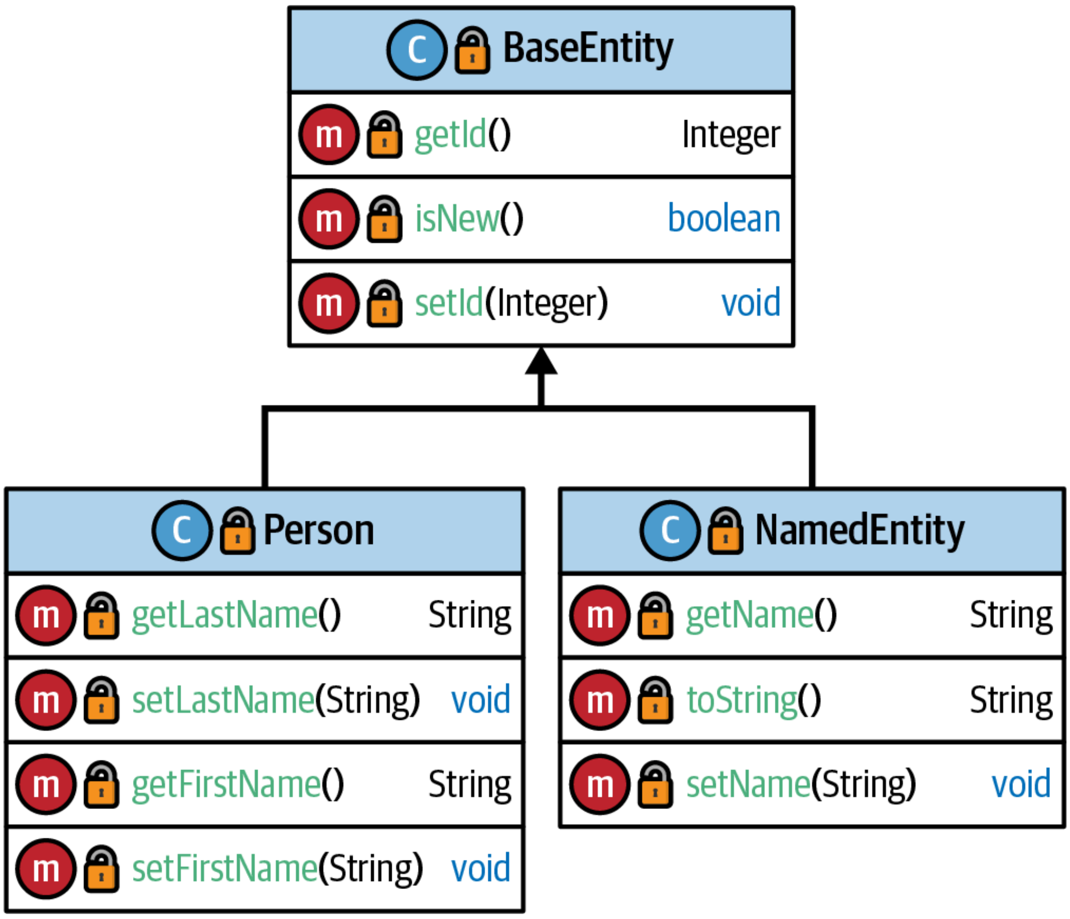

Figure 4-4. Sample class diagram

Sequence Diagrams

Sequence diagrams show a sequence of interactions, though they rarely show every single entity or every single interaction. Typically, they show only the entities involved in a particular flow. Most systems have nearly limitless interactions and as such, there is no reasonable way to document every single entity and every single flow. Sequence diagrams are frequently used to document the most interesting or architecturally significant interactions or as a way of showing a particular pattern or a standard use of a library.

Sequence diagrams may show operations, including parameters and return types. While this can be helpful to developers, it's also important to understand they can quickly get out-of-date with the code. It is possible to extract sequence diagrams from existing code, which can be helpful when starting on a new codebase. These diagrams are usually built by developers or solution architects. The audience is typically technical, including developers, architects, and DevOps. QA may also use sequence diagrams to help them understand what to test.

Sequence diagrams are used throughout a project. Early in a project, they define a pattern, but they are incredibly useful in knowledge transfer and onboarding. For example, the sequence diagram in Figure 4-5 returns to the API service, showing that a search determines whether the user is authorized to perform that function, and if so, how that request flows to the data store and back.

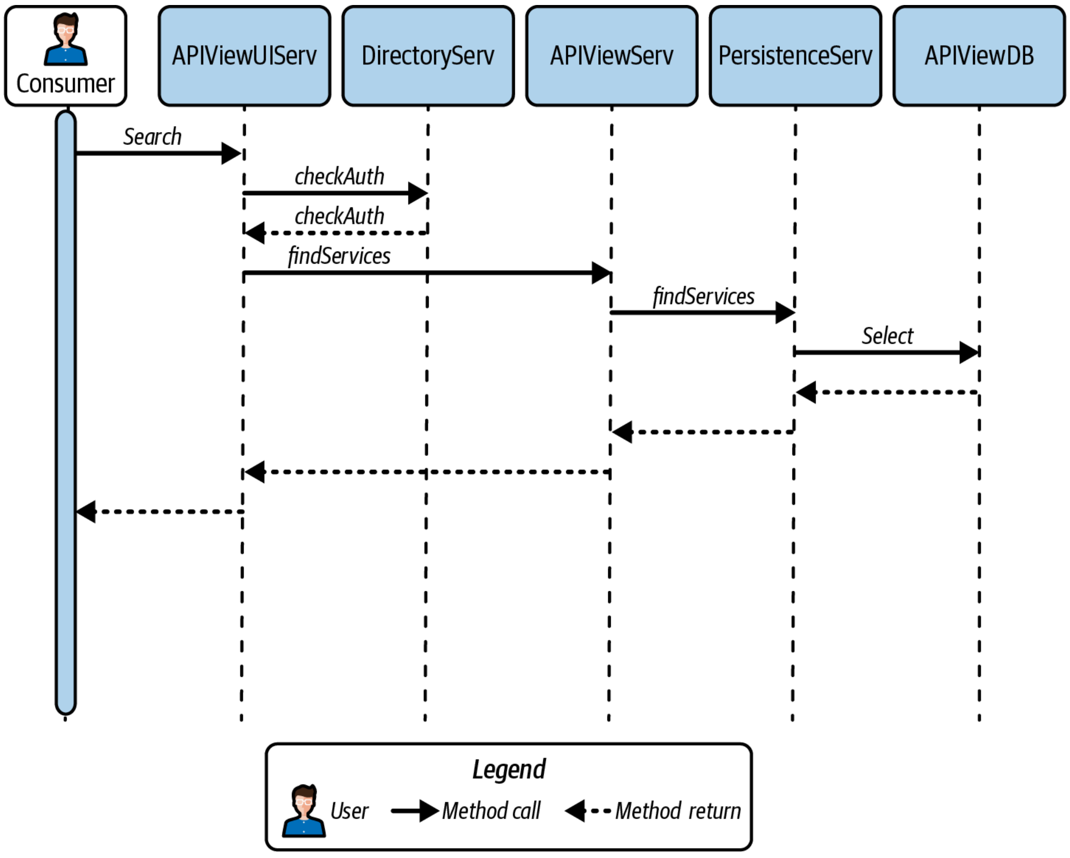

Figure 4-5. Sample sequence diagram

Deployment Diagrams

Deployment diagrams provide a runtime view of your system. They show the physical hardware nodes as well as the software that is running on them. Often, they show how the hardware is connected and show protocols as well as cardinality. Deployment diagrams are intended for a more technical audience, such as architects, developers, DevOps, and production support as well as infrastructure, architects, security, and your middleware team.

Deployment diagrams can be logical or physical and should be labeled appropriately. For example, a logical deployment diagram might represent a cluster as a single entity, while a physical deployment diagram would show the number of a given load balancer or server running at a given time in said cluster.

Deployment diagrams are used early on in a project to validate quality attributes. Systems that require 24/7 support will have a very different deployment than those that have less stringent uptime requirements. Deployment diagrams can help us understand whether an application will meet our scaling needs as well as validate business continuity. They also can help us find a more cost-effective deployment.

Many organizations will have a standard template for their deployments. Most companies will have standard cloud environments or on-premises approaches. In many ways, deployment diagrams are a bit like Lego blocks. Your company may have a defined set of tools and technologies that you can use, and you will mostly snap them together. Many organizations have standardized reference architectures that describe typical applications while at the same time putting some boundaries on your deployment options. Very few companies provide an unlimited toolbox. If your company has standardized on Amazon Web Services (AWS), you wouldn't create a diagram full of Azure-specific entities, and if PostgreSQL was your corporate-approved relational database, you wouldn't want to model a deployment using MariaDB. For an example deployment diagram, see Figure 4-6.

This example explores the self-driving car, showing which parts of the application are deployed within the secure zone versus which aspects live outside the firewall. These diagrams can also show specific versions of components.

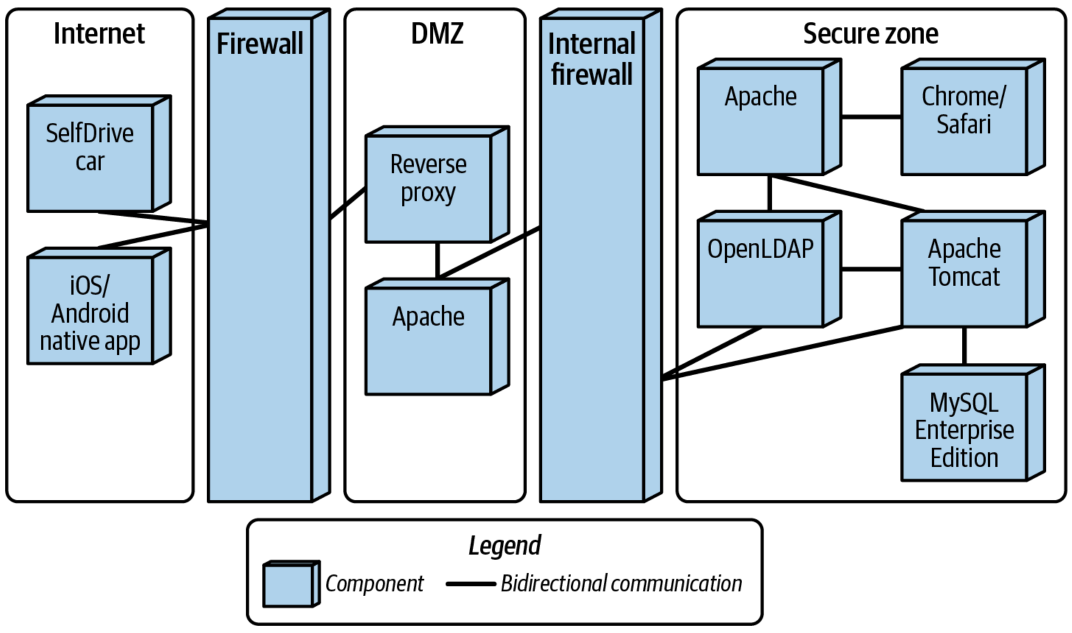

Figure 4-6. Sample deployment diagram

You may encounter a specialization of a deployment diagram known as a *security diagram*. This more detailed model describes the security mechanisms of an application. Security diagrams often include protocols and can leverage a deployment or technology view of the system. These are intended for a more technical audience, such as developers, architects, and security professionals.

Security diagrams are used throughout a project. They can define a pattern as well as validate that the solution meets the security needs of the project. It is important to consider personally identifiable information when interacting with a system as well as any regulations or laws that may apply to your system. For example, see Figure 4-7.

Once again, this example explores the self-driving car, showing which parts of the application are deployed within the secure zone versus which aspects live outside the firewall and must be treated as such. Notice the addition of protocols like HTTPS and LDAP.

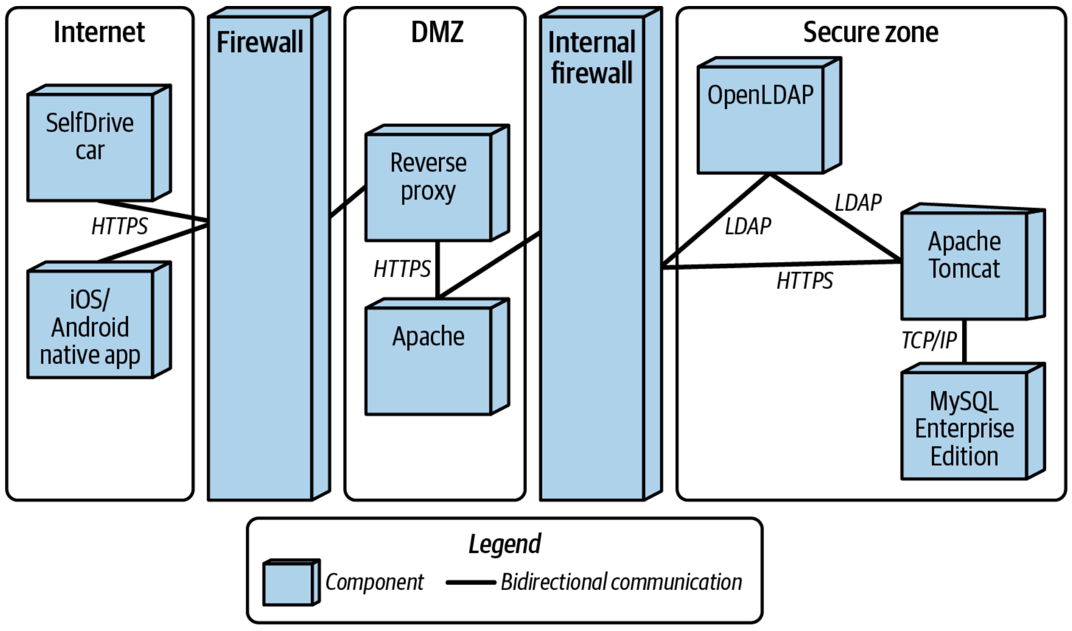

Figure 4-7. Sample security diagram

Data Models

Data models show data entities as well as relationships. They can be at different levels of granularity, from more conceptual to logical, all the way to the physical layout of the datastore. Essentially, they progress from high-level to concrete implementation. Conceptual models are very high level and are not normalized.[8] Logical models show business terms, often in a normal form. Physical data models show implementation details including data types. Again, they progress from less detail to more.

The audience for data models can range from customers to information architects to database administrators, as well as software architects, developers, and support personnel. Data models are often created very early on in a project to illustrate the domain, though they are often refined throughout the lifetime of a project. For example, Figure 4-8 shows some of the data entities from the API system. A given technology has one to many versions, and a vendor has one to many technologies and platforms.

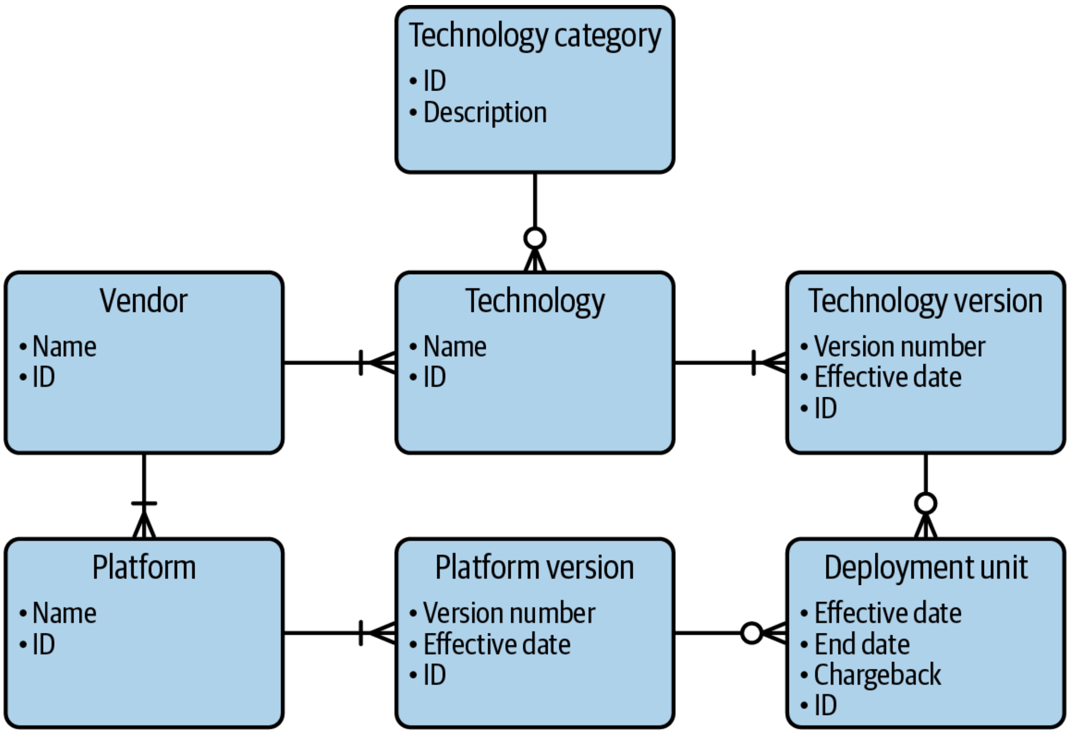

Figure 4-8. Sample entity–relationship diagram

Additional Diagrams

You might create any number of other diagram-like artifacts during the lifespan of a project. Again, in most instances, the creation process can be invaluable to understanding your domain, a specific problem, or a tricky interaction. You will often discover that people don't have a shared understanding of the problem. In no particular order, here are additional diagrams you might use:

Event storming
: A workshop-based technique for exploring a business domain. It can be done in person or virtually but involves business and technical stakeholders outlining domain events and the command that creates said domain events as well as the actor that executes the command. Different-colored sticky notes represent the various categories. Event storming is a very flexible and lightweight process that can be used in a variety of ways.

Value stream mapping
: A way of analyzing the current state of a system as well as modeling a future end state. It can be done in person or virtually. Value stream mapping allows you to see where there is friction or bottlenecks in a process.

User story mapping
: Uses sticky notes along with rough sketches to map out the user experience of a software system. It can be done in person or virtually and involves business and technical stakeholders working together to explore the system. High-level tasks are represented as activities, with the requisite steps and details laid out below.

Disaster recovery models
: Provide a detailed view of a system and help define the business continuity requirements for your project. It is important to consider the number of people impacted by an outage as well as the cost of downtime. These models show runtime entities as well as the hardware they are running on. They are used throughout a project lifecycle, though they increase in importance as you get closer to full production releases.

Modeling Best Practices

As discussed, models can be incredibly useful to communicate technical intent. An informal diagram is relatively quick to draft, which can be both a blessing and a curse! Diagrams can get out of hand. The following are a few practices that can help you rein them in.

Keep It Simple

Diagrams can be very noisy. It is easy to create a diagram for shock and awe purposes, an effort to overwhelm and bewilder an audience. Some diagrams are so crammed with text that they make you wonder if you need to get your eyes checked. Others have distracting visuals, with lines going everywhere and elements of all shapes and sizes. These diagrams show so much, they ultimately don't communicate anything but confusion.

Though common, these diagrams are rarely useful. Again, *the point of a diagram is to help you communicate something to others. If someone can't decipher your diagram, it's useless*. As your application grows, you will have more and more entities and more and more diagrams. Should you try to show everything on one diagram? While it can be useful to have an overview of your application, these diagrams tend to be very challenging to consume. If someone were to print it out, would it be legible?[9]

There is no rule to include everything on one and only one diagram. In many cases, you are better off breaking a diagram apart to make the individual diagrams easier to consume. One diagram can easily point to another diagram. Take a component diagram as an example. Were you to show every single component of your application, the diagram would quickly become unusable. However, you can typically break the diagram apart at logical points and have one diagram refer to another. You may still want to have an overview diagram that shows all of the components in one view, but it should rarely be your only model of the system.

Know Your Audience

Much like writing code (see Chapter 3), the best software engineers focus on the person consuming their diagrams. A highly technical diagram that's perfect for a developer might not work so well if you're showing it to a nontechnical person.

Diagrams Require Context

Nate here. Many years back, I was the lead developer on a new web application. As part of my work on this project, I was very proud of the class diagram I had created—despite its complexity, it had no crossing lines! I scheduled time with my architect to review my work. I proudly passed my diagram across the table for his review, only to have him shrug and pass it back. I'll admit I was a little heartbroken; after all, there were no crossing lines. While I wasn't expecting a cookie for my efforts, I certainly expected some praise.[10]

What I didn't understand at the time is that my architect did not have enough context to understand my class diagram. While I had hoped he would appreciate my craftsmanship, he had no way of identifying mistakes that weren't clear and obvious. He did not understand our business domain to the level that I did on that project and thus couldn't effectively comment on it. He had to trust that my work was correct. That isn't to say these diagrams aren't helpful, just that you must understand their limits.

Be Careful with Your Color Choices

You will often use color to designate certain things in your diagrams. For instance, new, modified, or untouched components may all have different colors. If you are going to use color, be sure to include a key identifying your color choices. Just because your team uses green for *new*, do not assume that everyone else does as well.

Even if you haven't printed anything in years, you should assume that someone in your organization will at least attempt to print your diagram. A diagram may look fantastic in its chromatic splendor on your monitor; however, once it's printed out in grayscale, it may lose much of its meaning. You should also expect that someone in your audience is color blind. The moral of the story: have an alternative to colors.

Establish Standards and Templates

It can be very helpful to define a set of standards within your organization. A set of diagrams that you routinely use along with a set of colors that you expect will give you consistency across projects. Just don't be overly prescriptive. Allow room to add or remove diagrams as required on a given project.

Do I Really Need This?

Nate here. Years ago, I worked in an organization with a particularly prescriptive approach to project documentation. As an architect, I was expected to create any number of diagrams; however, I was quick to throw out things I did not need. For example, our template at the time required me to include a use case diagram. Considering this was an Agile project using stories and not use cases, I thought it was odd to create a diagram just for the sake of creating a diagram. So I didn't. Don't be afraid to push back on a template.

Tools

Your organization probably already has a modeling tool or two available to you. However, you can produce perfectly suitable diagrams without instigating a lengthy procurement process.

If your company doesn't have anything, consider the following (nonexhaustive list):

- OmniGraffle is a diagramming tool for MacOS and iOS from the Omni Group. It includes dozens and dozens of stencils that allow you to quickly create standard software diagrams. With a WYSIWYG drag-and-drop environment, the learning curve is relatively low.

- Microsoft Visio is a diagramming tool for Windows. Visio allows you to create various diagrams, from flow charts to mind maps, as well as UML diagrams. Like OmniGraffle, Visio uses a WYSIWYG drag-and-drop environment, making it very easy to pick up. If your work laptop includes the Microsoft Office suite, odds are you've already got Visio installed. There are several downloadable stencils specific to software engineering and public cloud providers such as Azure, AWS, and GCP.

- Mural, Miro, and Lucidchart are browser-based tools that act as a distributed collaborative whiteboard. They allow you to create any number of diagrams. Multiple people can interact with the same canvas simultaneously, making them ideal for collaborative sessions.

- Mermaid is a JavaScript-based diagramming and charting tool that uses Markdown-inspired text definitions to create and modify diagrams dynamically. It's particularly popular for embedding diagrams in documentation, GitHub wikis, and other platforms that support Markdown. Mermaid can create flowcharts, sequence diagrams, class diagrams, state diagrams, and more using simple text syntax.

- Structurizr follows the diagram-as-code model, allowing you to create any number of diagrams using the Structurizr DSL. Though designed to support the C4 model, it is not limited to those visualizations and includes themes for common cloud-based architectures.

- PlantUML supports many common UML diagrams, allowing you to generate models from text files via a command-line or GUI interface.

- Your IDE. Odds are, your IDE natively supports directly creating models as well as extracting them from your codebase. You may need to add a plug-in or two, but today's editors often include serviceable modeling tools.

- Presentation software such as PowerPoint, Keynote, and Google Slides are also incredibly commonly used, largely because of their ubiquity within most enterprises. Though not designed as modeling tools, stencils specific to software engineering can supplement the basic shapes included by default. They can also facilitate collaborative sessions, allowing multiple people to work together on a diagram.

- AI tools are becoming more common, and many are capable of extracting diagrams from a codebase. Hallucinations are always possible, though, so you should carefully review the output.

When in doubt, choose a tool that everybody has access and/or a license to use.

Generating Code from Models

Many years ago, some modeling tools would allow you to generate code from diagrams. While this could be helpful when starting a project, these approaches have largely been consigned to the dustbin of software history.[11] That's because these tools had a major flaw: unless developers carefully observed the rules of the tool, any code they added to the generated files would often get deleted the next time code was generated from the model.

Some tools worked bidirectionally, at least in theory—modifying the models as you updated the code, and vice versa. Unfortunately, they didn't result in the huge productivity gains they promised.

Regardless of what tools you employ, don't forget to version control your diagrams. While the diagrams-as-code tools are a more natural fit for versioning, modern versioning tools can handle visual elements fairly well.

While more complex tools often include powerful features, they usually come with steep price tags as well as a lengthy learning curve. Regular use can remedy the latter, but these tools can make things more complicated than is strictly necessary. Do not be afraid to use a simpler tool if it allows you to get your message across more easily and more quickly.

However, understand that the message you are trying to get across with your diagram is the most important part of the equation. The tool is merely there to help you express that intent. If you like a tool and it helps you be more productive, by all means you should use it. However, don’t let the tool hold you back. Paper and pencil make for excellent modeling tools as do whiteboards. When in doubt, keep it simple.

Remember, diagrams are not a substitute for working code. They can be very helpful, but they can also be out-of-date with the code. Some diagrams can be extracted from the code itself, either on demand or as part of a build pipeline, which can alleviate the out-of-sync issue. However, you should always consider the code the ultimate source of truth.

Wrapping Up

Models and diagrams are an important part of a software engineer’s toolkit. While code is the ultimate source of truth, diagrams can help you communicate key concepts to technical and nontechnical stakeholders; the challenge is knowing when to use a particular approach and when to skip it. Use modeling tools when and where they make sense.

Putting It into Practice

Pick an interesting flow in an application you know well, or if you’re feeling brave, an open source library you’re familiar with. Draw a sequence diagram or two. Ask a colleague for some feedback.

Extract a class diagram from your application. What surprises you about the relationships? What insights does the resulting model give you about your application?

Ask your architect to share any diagrams they’ve created for your systems. Spend some time getting comfortable with them. If you have any questions about them, ask! What would you do differently if you were asked to create one of those diagrams? If something isn’t clear to you, discuss it with your architect. Some projects generate diagrams as part of the CI/CD pipeline; if yours doesn’t, it may be worth adding.

Additional Resources

- The C4 model for visualizing software architecture
- *User Story Mapping* by Jeff Patton (O’Reilly, 2014)
- *Communication Patterns* by Jacqui Read (O’Reilly, 2023)
- *Creating Software with Modern Diagramming Techniques* by Ashley Peacock (Pragmatic Programmers, 2023)

- *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, 3rd Edition, by Martin Fowler (Addison-Wesley Professional, 2003)

[1] As well as terms such as *architect* and *quality assurance*, and within the data domain you'll encounter librarians, scientists, and ontologists.

[2] Though it can be more difficult and expensive than some assume.

[3] If they are sufficiently out-of-date, one can argue the diagrams are actively harmful.

[4] It is also possible to generate code from models, something that you may encounter primarily in safety-critical systems.

[5] To save you a web search, *Z notation* is a formal specification language developed in the late 1970s and is based on mathematics.

[6] It has to do with cascading deletes, but you'll have to look up the nuance.

[7] *Cardinality* means the number of elements in a set.

[8] *Normalization* is a process that reduces redundancy and improves data integrity.

[9] Trust us, there's always someone who will want a print copy.

[10] OK, maybe I was expecting a cookie.

[11] Today developers just vibe code solutions, amirite?