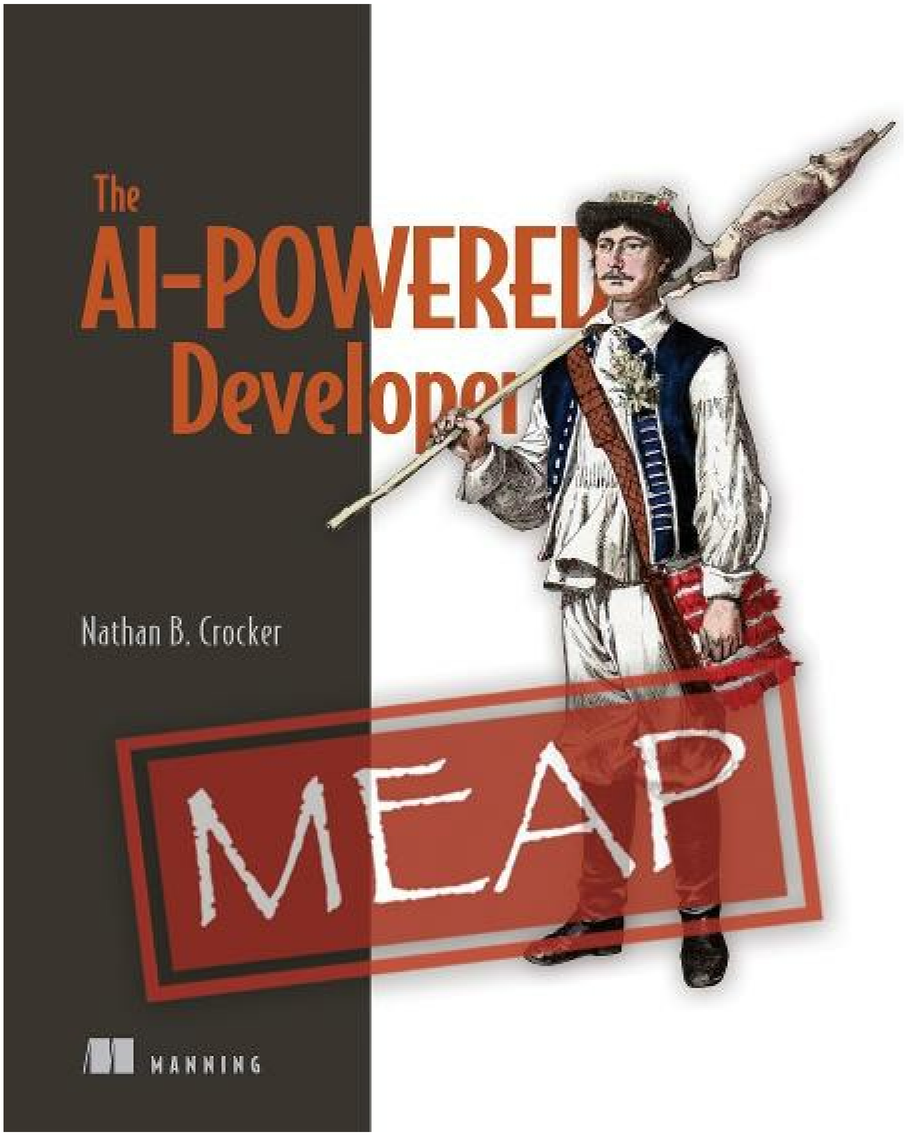
The AI-POWERED Developer

Nathan B. Crocker

MEAP

MANNING



The AI-POWERED Developer

Nathan B. Crocker

MEAP

MANNING

The AI-Powered Developer MEAP v02

1. Copyright_2023_Manning_Publications
2. welcome
3. 1_Understanding_Large_Language_Models
4. 2_Getting_Started_with_Large_Language_Models
5. 3_Designing,_Then_Building
6. 4_Building

MEAP Edition

Manning Early Access Program

The AI-powered Developer

Version 2

Copyright 2023 Manning Publications

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes.

These will be cleaned up during production of the book by copyeditors and proofreaders.

https://livebook.manning.com/book/the-ai-powered-developer/discussion

For more information on this and other Manning titles go to

manning.com

welcome

Thank you for securing your copy of the MEAP for *The AI-powered Developer*.

This book is designed to transition readers from having little to no experience with AI-powered coding tools to becoming proficient developers able to leverage these powerful technologies in their projects. Given the broadness of introducing both programming and AI concepts, we assume you're familiar with Python.

My journey to writing this book has been enriched with diverse experiences. Working with various projects, I've faced the task of integrating and leveraging AI tools in development workflows. Back then, the robustness of AI assistance and documentation we enjoy today was not available, making the transition challenging yet incredibly rewarding.

Throughout this book, we'll delve into the core aspects of generative AI models like ChatGPT and GitHub Copilot, providing a hands-on understanding of their application in real-world scenarios. We'll explore how these AI models can interact with Python to create efficient, self-documenting code, make predictive suggestions, and assist in the design phase.

It's essential for professionals to thoroughly understand their tools, and this book ensures just that. We will familiarize you with various features of AI models, ranging from beginner-friendly to advanced capabilities. Additionally, we will also discuss how these AI models fit into the testing cycle, their role in debugging, and their ability to explain code and concepts.

The integration of AI tools like ChatGPT and GitHub Copilot into coding has introduced revolutionary changes and possibilities in the software development community. “The AI-powered Developer" aims to empower you to take full advantage of these opportunities, covering a broad range of topics to ensure that you can create the applications you envision while, or even after, reading the book.

Your feedback is crucial for crafting the best possible book, so I encourage you to share your comments and questions in the liveBook Discussion forum. I'm eager to know if my explanations are resonating with you and aiding in your understanding!

-Nathan B Crocker

In this book

Copyright 2023 Manning Publications welcome brief contents 1 Understanding Large Language Models 2 Getting Started with Large Language Models 3 Designing, Then Building 4 Building

1 Understanding Large Language Models

This chapter covers

- Introducing Generative AI (Specifically Large Language Models)
- History of Generative AI
- Exploring the benefits of Generative AI
- Determining when and when not to use Generative AI

Whether you realized it or not, and whether you want to admit it or not, you have quietly received a promotion. In fact, every professional software engineer has. Almost overnight, we have gone from staff engineers to engineering managers. You now have the world's smartest and most talented junior developer on your team. Guiding, mentoring, and performing code reviews should become part of your daily routine. You now have Generative AI as your new coding partner. This chapter will provide you with an overview a subset of Generative AIs called Large Language Models (LLM), specifically Chat GPT, GitHub Copilot, and AWS CodeWhisperer.

1.1 An introduction to Large Language Models

Generative Ais, specifically Large Language Models (LLMs) are radically transforming how we think about and develop software. Rather than designing and coding out modules, components, and tests, we will describe the software that we want these Ais to build, and they will generate the body of this work for us. This is a natural trend in the fields of software engineering: our compliers have gotten smarter (Rust's compiler being a prime example, which eliminates an entire category of bugs), as has our tooling (IntelliSense in source code), and our programming languages have become more expressive and more productive. While this might make these Generative Ais seem more evolutionary, than a revolutionary; they are, in a sense, both.

This book will examine, compare, and contrast three such Large Language Models: GitHub's Copilot, OpenAI's ChatGPT, and Amazon's CodeWhisperer. The latter will receive the least coverage as it is largely analogous with Copilot but is more useful of and focused on development within the AWS eco-system.

GitHub Copilot and ChatGPT both use the GPT-4 Codex model created by OpenAI behind the scenes. Microsoft licensed the software from OpenAI, using the source code within the public repositories within GitHub (which Microsoft owns). GitHub built a service that will take the context provided by your Integrated Development Environment (IDE), such as Visual Studio Code or IntelliJ, and send that context to the GitHub Copilot service. This Service will use the OpenAI Codex to generate up to ten possible solutions, given the context that you have provided via the comments and code in your file. Codex attempts to match this context against examples that it finds in the corpus of its training data. These code solutions will be returned to your Integrated Development Environment for you to select from. You review all of the code suggestions and accept the one that is closest to your intent. The supervision that you provide here is very important: it is not uncommon for the solutions to be out of date or inexact. Should you accept one the solutions, then your "accepted" solution is then sent back to the GitHub Copilot Service to enhance the model further and thus the suggestions.

Microsoft is betting big on this technology with Copilot. As is Google with Bard. And it is easy to see why.

This book teaches you how to manage Generative Ais such as GitHub Copilot. Trivial examples will give way to incredibly complex ones that will leave you stunned. This book assumes you have little knowledge of *using* Generative AIs. You have likely heard about Generative AIs for some time. You have probably thought the concept is exciting and worth looking into Generative AI at some point. Well, there is no time like the present. This book will take you through the basics, from setting it up in an Integrated Development Environment to using it to 10x your productivity, output, and hopefully enjoyment of coding.

Over the course of this book, you will see example after example of how one Generative AI is better suited for a given task. This will help you build intuition around when you would want to use one or the other, as well as when you might want to avoid them all. Let us take a brief survey of each of the Generative AI’s core strengths.

ChatGPT excels at generating responses that mimic human speech and written language. It is therefore very good at documentation and commenting in code. Because of its ability to process Nature Languages (NLP), it can also perform the reverse: summarize text and capture the sentiment. You can also use it to improve these areas: have it rewrite or rephrase copy.

ChatGPT can generate code snippets, functions, applications, and whole chatbots. In addition, you can use it to autogenerate tests. We will do all of these things in subsequent chapters.

GitHub Copilot (and CodeWhisperer) confer the following benefits to developers: they assist with code completion, fixing errors, and refactoring. This will make developers more productive, in terms of output (lines of code per time period), but they can also automate repetitive tasks. Given that Copilot’s training data was collected by culling public repositories, the developer will have suggestions to increase the overall code quality.

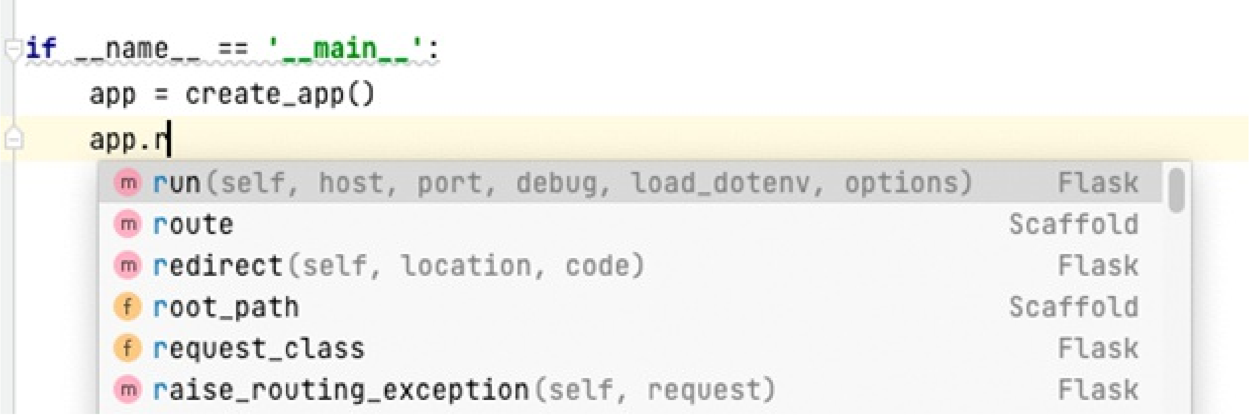
Copilot can also assist in better understanding of a foreign code base. It will provide suggestions as to how to navigate this code base, as it can help one better understand the relationships amongst the classes and code.

As you use these tools, you will notice that your velocity changes dramatically as you better understand the capabilities and limitations of your new programming partner. Your programming partner will also get better at working with you since it can “remember” your programming style and approach. Working with generative Ais will allow you to tackle much more complex problems. You will write better, cleaner code, with fewer bugs. All while moving faster than you thought possible. Sound like a dream, or worse, fluff? It isn’t.

One might ask themselves Isn’t this just a better version of IntelliSense? You might ask yourself this question after looking over the first, few examples; however, by the end of the next chapter, having used Generative AI, you will be able to appreciate the differences.

In figure 1.1, you will see Microsoft Visual Studio Code provide an IntelliSense auto-completion suggestion to start the Flask application. Note that this is inline, and the suggestion comes as I edit the code.

Figure 1.1 IntelliSense auto-completing the Flask run method.

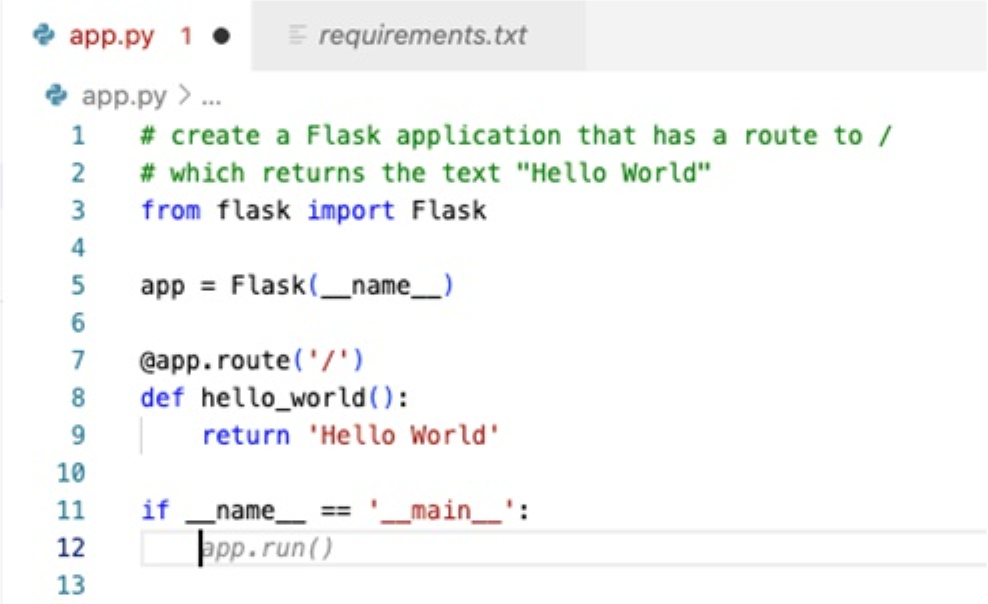


```
if __name__ == '__main__':
    app = create_app()
    app.r
```

m run(self, host, port, debug, load_dotenv, options) Flask
m route Scaffold
m redirect(self, location, code) Flask
f root_path Scaffold
f request_class Flask
m raise_routing_exception(self, request) Flask

In figure 1.2, notice that GitHub Copilot has made the same suggestion based off of the method name and signature. That is, it wrote the code before I started writing the implementation. The excitement behind generative Ais is tied to this power: its predictive nature. As the prompts get more explicit, the suggestions get more exact. We will explore this further in later chapters.

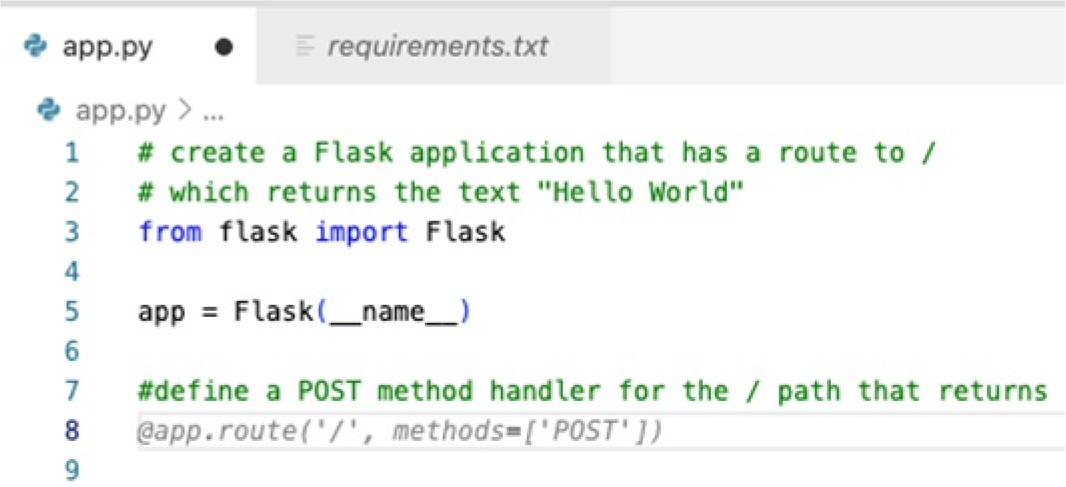
Figure 1.2 GitHub Copilot solution as to how to run the Flask application.



```
# create a Flask application that has a route to /
# which returns the text "Hello World"
from flask import Flask

app = Flask(__name__)

@app.route('/')
def hello_world():
    return 'Hello World'

if __name__ == '__main__':
    app.run()

```

Figure 1.2 presents a trivial example and does not make a compelling case for why one would want to use Generative Ai. However, in this same Flask application, what if you need to create an end point that can handle the input from a POST method, but you forgot the syntax? Would we need to open the official documentation and try to find how to do it? No, we could just ask GitHub Copilot.

Figure 1.3 GitHub Copilot generating a POST method handler



```
# create a Flask application that has a route to /
# which returns the text "Hello World"
from flask import Flask

app = Flask(__name__)

#define a POST method handler for the / path that returns
@app.route('/', methods=['POST'])

```

You can see that Copilot offered several albeit similar suggestions as to how to complete this code. Declaring the method, would have gotten the first suggestion auto-completed inline in our IDE. No need to stop and use the mouse. This approach keeps you in the code and in the flow state for longer without unnecessary distraction. Now, if only Copilot could fetch us a coffee…

1.2 History of Generative AI

It is worth taking a quick detour to understand a little bit about the genesis of the technologies that we study throughout the next few chapters.

Generative AIs are a sub-set of Artificial Intelligence. Artificial Intelligence has been around and actively researched for more than sixty years. The Logic Theorist is considered the first application of artificial intelligence, predating the term “artificial intelligence”. The Logic Theorist was the brainchild of

Herbert Simon and Allen Newell, with some contributions by Cliff Shaw. Simon and Newell were attempting to teach a computer to think.

While this attempt did not result in a genuinely thinking machine, the Logic Theorist was able to produce better, more detailed mathematical proofs than contemporary mathematicians Alfred North Whitehead and Bertrand Russell. The peculation and theorizing about what Logic Theorist would do to the field of mathematics mirrors what we are seeing in the news surrounding ChatGPT today.

The term “artificial intelligence” would not exist until the RAND Corporation hosted the Dartmouth Summer Research Project on Artificial Intelligence in 1956. Then, prominent researcher John McCarthy (the original author of Lisp) and computer scientist coined the term “artificial intelligence,” unveiling it at this conference.

Research into AI continued for decades, incrementally improving with public interest waxing and waning. IBM’s Garry Kasparov was defeated by Deep Blue beat at chess in 1997. Ken Jennings lost to IBM’s Watson at Jeopardy! in 2011. A few years later, in 2014, generative adversarial networks (GANs) were invented. The advent of GANs kicked off renewed interest in AI, as it was able to create realistic images and so-called deep fakes.

In 2017, a new type of deep learning model was invented: the transformer. This laid the groundwork for the Large Language Model (LLM): LLMs such as GPT-3. Whereas GANs could generate images from noise maps, LLMs could create text and software. Around this time, the company OpenAI – the stewards of ChatGPT – was founded.

But why was OpenAI founded? What was their goal? Let’s ask ChatGPT.

Q: what is the history of openai?

A: OpenAI is a research company that was founded in December 2015 by Elon Musk, Sam Altman, Greg Brockman, Ilya Sutskever, Wojciech Zaremba and John Schulman. The company's mission is to develop and promote friendly AI in a way that benefits humanity as a whole. OpenAI initially focused on developing and promoting AI in the gaming and robotics industries but has since expanded its focus to include a wide range of industries. In 2019, OpenAI released GPT-2, a powerful language processing AI model, and in 2020 GPT-3, which has been widely adopted in natural language processing tasks and has become one of the biggest AI models to date. In 2023, we saw the release of GPT-4, the most advanced model to date. It has the ability to connect to the internet amongst other vast improvements. OpenAI has also been involved in numerous research projects in the fields of AI safety, robotics, and machine learning, and has established partnerships with a number of organizations and companies to advance AI research and development.

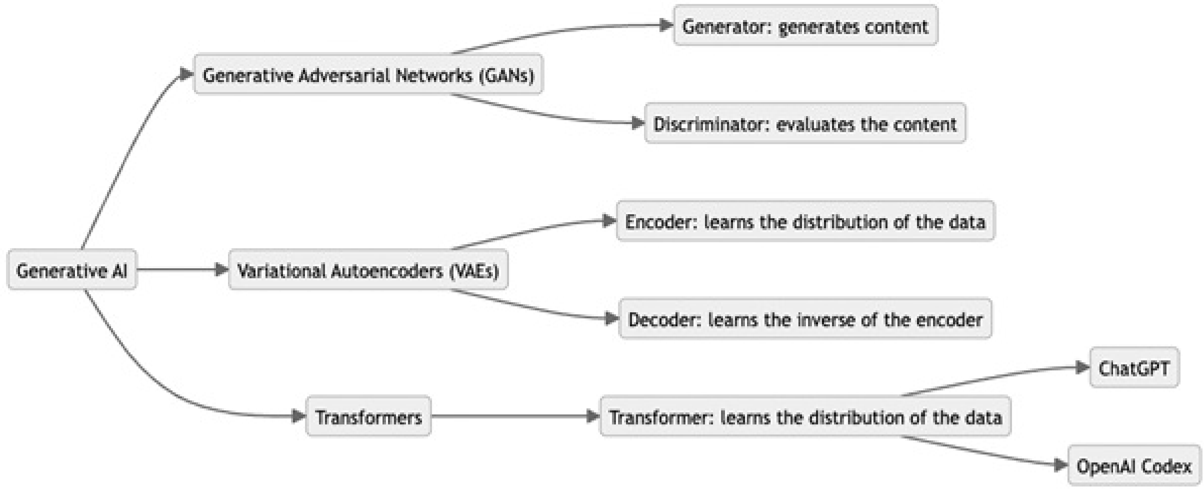
A key detail buried in this text is that there are companies that are actively working on making AI systems easier to use. Given that this trend is unlikely to end (and will likely only intensify in the future) does it not make sense to get in early? You will stay ahead of the curve. This will make your skills as an AI assisted developer all the more valuable.

1.3 Grokking Generative AI

Generative AIs are a subset of artificial intelligence. They are traded on a large dataset to learn the patterns and structures of the data. Once trained, they will use this data to generate new data similar to the dataset that they were trained on in terms of this new data’s structure. This is the generative part in the name Generative AI.

There are three main types of Generative AIs: Generative Adversarial Networks (GANs), Variational Autoencoders (VAEs), and transformer-based language models. ChatGPT and OpenAI’s Codex are examples of the latter. We will briefly walk through how each of these types of Generative AIs function.

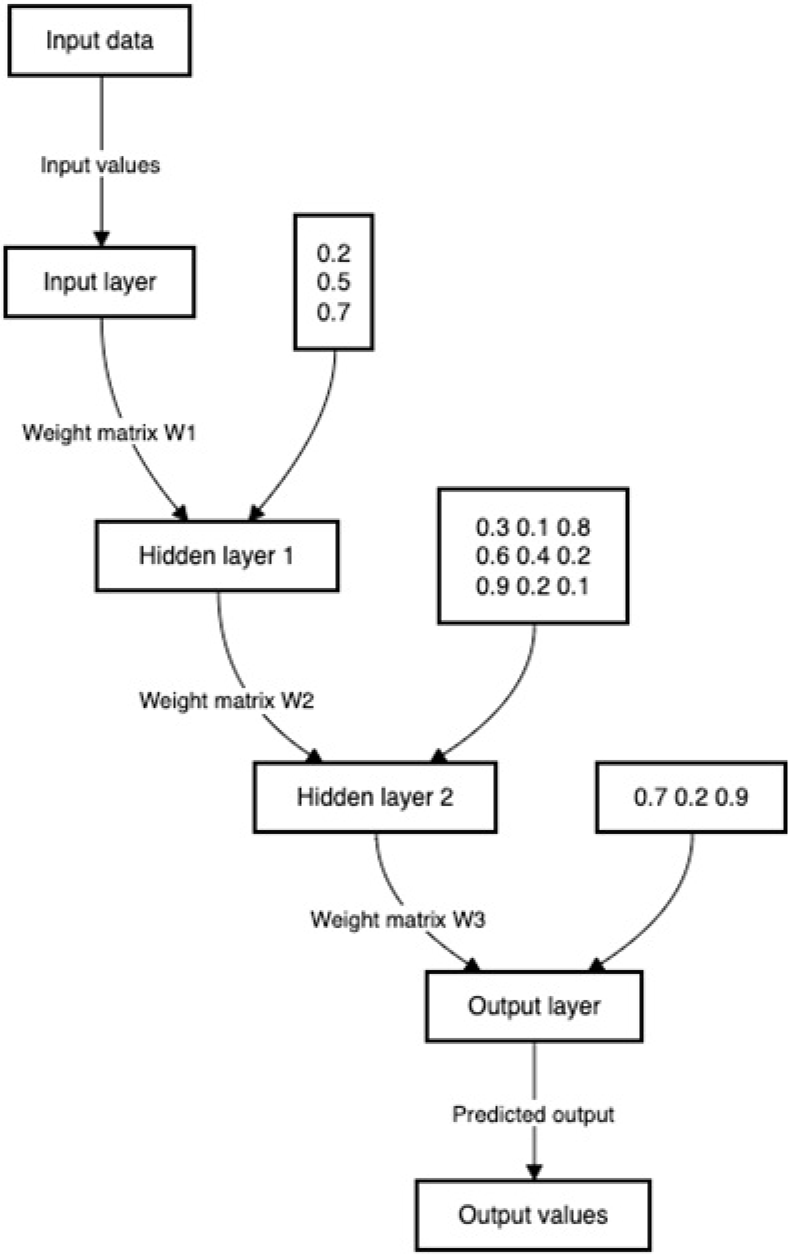
Figure 1.4 There are three main types of Generative AI: Generative Models, Generative Adversarial Networks, and transformer based.


Generative AI
Generative Adversarial Networks (GANs)
Generator: generates content
Discriminator: evaluates the content
Variational Autoencoders (VAEs)
Encoder: learns the distribution of the data
Decoder: learns the inverse of the encoder
Transformers
Transformer: learns the distribution of the data
ChatGPT
OpenAI Codex

Many Generative AIs employ a neural network to create output; be that text or code generation or images. A neural network is patterned after how humans' minds work, as neurons pass signals to one another. You can visualize this as a directed graph in which data that exceeds certain thresholds are passed to the next node in the graph.

Data is encoding the input layer, which is called the outer layer. The output layer is connected to hidden layers. Behind the hidden layers are other, numerous hidden layers through which the data must traverse. All of the nodes in the neural network are connected via calculated numerical values, representing the strength of the connection between the neurons using back-propagation (represented as lines in Figure 1.5), which have thresholds that must be exceeded in order for the data activate the next layer. If the data makes it to the output layer, then the data is returned from the network. There is no guarantee that data will be returned, however. The data might be filtered out.

Figure 1.5 Visualizing a neural network. A very tiny one. Outer nodes are exposed so that they can accept input. As Hidden nodes are traversed these inputs are either discarded or forwarded to the next node. If input makes to the Output nodes, then it is returned.



A Generative AI uses very large data sets to train these models. In the case of GitHub's Copilot, this large data set was the contents of the publicly accessible repositories within GitHub itself. If you have ever contributed to an open-source project, then you might have code from which Copilot has trained.

While many Generative AIs use a neural network, how they use it determines the AI's type. A Generative Adversarial Network (GAN) use two neural networks: one called generator and one called the discriminator. The generator network generates fake data based off of the training data set. The discriminator tries to identify fake data. These networks are adversarial in nature as the generator attempts to create data that is indistinguishable from the real data and the discriminator attempts to discern if the data is real or fake.

Variational Autoencoders (VAEs) use two networks are well: one for encoding and one for decoding. In one sense the encoding network simplifies the input by reducing the data into a lower-dimensional representation. The decoding network then maps this lower-dimensional representation back to the original data space. The whole point of this is to be able to generate new data through sampling.

The final type is *transformer-based models*. The transformer model is a type of *feedforward* neural network that uses *self-attention mechanisms* to process sequential data, such as natural language text. During training, the weights of the network are adjusted to minimize a loss function, such as cross-entropy.

In a feedforward neural network, the input flows in one direction, from input layer to output layer, with no feedback connections between the layers. Additionally, no information or error signal flows back from the output to the input layer. Therefore, the neural network's output is determined solely by the input data and the weights assigned to the connections between the layers.

Self-attention mechanisms allow the network to selectively attend to different parts of the input sequence based on their relevance to the current output. In a transformer, the input sequence is first embedded into a vector space using an embedding layer. The embedded input sequence is then fed into an encoder network, which consists of multiple layers of feedforward neural networks. Each encoder layer applies self-attention mechanisms to capture the relationships between the different parts of the input sequence.

The self-attention mechanism calculates an attention score for each part of the input sequence based on its relationship to the other parts of the sequence. These attention scores are then used to weight the contributions of each part of the sequence to the final output of the encoder network. This allows the encoder network to selectively focus on the most important parts of the input sequence, while ignoring irrelevant parts.

The output of the encoder network is then fed into a decoder network, which also consists of multiple layers of feedforward neural networks. The decoder uses self-attention mechanisms to generate an output sequence based on the input sequence, one token at a time.

An analogy for the relationship between encoders and decoders in a transformer network is that of a compiler and linker. Just as a compiler breaks down high-level code into low-level instructions and a linker combines those instructions into a final executable program, the encoder network breaks down the input sequence into meaningful units and the decoder network combines those units into a final output sequence. The use of self-attention mechanisms in transformers is similar to the way a compiler and linker optimize code for better performance.

As previously stated, many Generative AIs use neural networks, but not all of them do. Some are rules-based, generating output by applying the rules to the inputs. Still others are evolutionary in nature, iterating on results, and selecting based off of the goodness of fit.

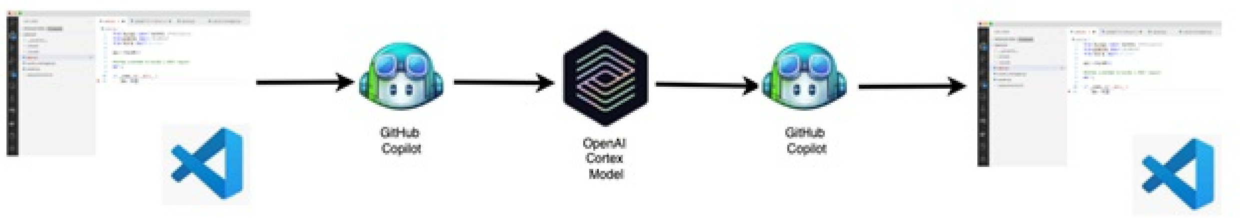
We can now walk through a descriptive example of how you would interact with Copilot. As you begin to type in your favorite IDE (VS Code in this example), the Copilot plugin will send your comments or code (sometime all it takes is a function’s name!) into the Copilot service.

This service turns your lines of code or comments into natural language prompts, which are then run through OpenAi’s Codex model. The model will generate suggestions based off of the training data set. GitHub refers to this as code synthesis. They claim that this training data set contains billions of lines of code from dozens of programming languages.

Once the top ten solutions are returned from the Codex model, the Copilot service will return these suggestions back to your editor. You select the suggestion that most accurately captures your intent or need. Or you reject all of the suggestions. Your selection is then returned to the Copilot service to better train the model.

Figure 1.6 Your code is sampled and encoded by the Copilot plug-in. It is then set to the OpenAI Codex model, where suggestions are generated. These suggestions are then returned to your VS

Code session.



GitHub is constantly improving their Copilot service. A recent release (as of December 2022), they boast that the acceptance rate is 46% across all programming languages and with Java specifically, it is 61% on average[1].

1.4 When to use and when to avoid Generative AI

Excitement around these technologies is growing. Since its public release in late November of 2022, there are hundreds (possibly thousands) of articles about the various dimensions of ChatGPT. Will it ruin education? Is education required anymore? Are software engineers necessary anymore?

It is easy to give into the pessimism. There are lots of unknowns and the full impact of this technology has yet to be revealed. However, you should form your own opinion as you work through this book. It is my hope that you will see the positives of generative AI and use them for good. You will use it to grow as a programmer. As you use them, you will become a better programmer.

One of the best ways to grow as a developer is to read good code. OpenAI has curated some of the best code on the planet; it is all at your fingertips. You also now have some of the worst code available at your fingertips as well. You can learn from good examples as well as bad. Being able to discern the difference is the key to growing.

So, when should you use generative Ais? Every opportunity that you can! (We will discuss some exceptions to their usage shortly.) It is fascinating to engage with generative AIs. You will learn how to use them better, find shortcuts, discover new features, and have stary-eyed child-like delight every moment of it.

While employing generative Ais in your daily job would appear to make a lot of sense (because it does), it should be noted that it is not pertinent to use in all cases.

If you were given a take home coding exam, should you use a generative AI to complete this exam? Unless it was explicitly stated that you can, then you should avoid it. It could be construed as cheating if the tester did not anticipate you using it.

Should they anticipate that you would use you? Yes, they probably should at this point. Further, one might argue that, given that the purpose of the exam is to assess the coding abilities of potential candidates, potential employers should try to construct real world conditions in order to best assess one’s ability. This should include all tools and resources that one would have available to them doing the course of their day. These tools would include generative Ais.

You should be especially careful of using them in an academic setting. Proper attribution is complex, at a minimum, and the line between inspiration and plagiarism with Generative AI is a very fine one. If your usage is determined to be plagiarism, the consequences will be dire and permanent: expulsion and you might be barred from future enrollment at other institutions. Proceed with extreme caution.

In general, use your better judgement. If there is a chance that you might run afoul of any copyright laws or administrative policy, then do not use it unless you are granted specific permission to do so. It would not hurt to talk to your corporate IT or InfoSec team at work, just to make certain that you comply with corporate policies related to corporate systems and computers.

One final note: Generative AIs are tools and like any tool, you need to have some idea of what you are doing. You should have some idea about what the correct answer should be. You should use them in domains, in which you have some idea what is going on. In cases like these, you will find yourself exploring the landscape, deepening your understanding of the domain as well as learning faster.

Now that we have explored an abridged history of Generative AI, seen some use cases for Generative AI, and applied some important guardrails, we will go hands-on in the next chapter, examining how to start the same project using these three Generative tools.

1.5 Summary

- Generative AIs are both evolutionary and revolutionary. Evolutionary in the sense that they are a just another iterating on the tools that we as developers use every day. Revolutionary in that they will transform how we do our jobs. In fact, they will change our jobs.
- The future of development will be the management of Generative AI. Even the mythical 10x developer will not have the productivity of a developer with an AI partner; an AI powered developer will produce higher quality code at a substantially faster rate, at lower cost than one who is not. We will spend more of our time training our AI partner to do what we want, how we want, then we will write code without the AI.
- While there are several Generative AI out in the wild, we will explore three of the most popular:
    - ChatGPT – Has been making headlines since November 2022.
    - GitHub Copilot – The most popular Generative AI that uses in an Integrated Development Environment. Financed and promoted by Microsoft.
    - Amazon Web Services CodeWhisperer – A product similar to Copilot but backed by Amazon.
- Some of the world's biggest companies are making a significant investment in Generative AIs (Microsoft, Amazon, Alphabet) and making them easier to use (Open AI).
- Generative AIs make use of extremely sophisticated neural networks, resembling ours, to filter and map input to new previously unseen output.
- You should check with your professor or teacher prior to making use of generative AIs for school work.

[1] Shuyin Zhao. "GitHub Copilot now has a better AI model and new capabilities." *GitHub Blog*. https://github.blog/2023-02-14-github-copilot-now-has-a-better-ai-model-and-new-capabilities/. Last accessed: Feb 14, 2023.

2 Getting Started with Large Language Models

This chapter covers

- Engaging with ChatGPT
- Learning the basics of using Copilot
- Learning the basics of using CodeWhisperer
- Contrasting the differences between these three Generative AI offerings

In this chapter, we will dive into using three of the significant Generative AIs. We will use them to build a model Information Technology Asset Management (ITAM) system. Commercial and Open Source ITAMs offer many functions related to the management of software and hardware of an organization. We will build out the hardware management component. Limiting the project's scope to just this one component will allow us to focus on how we engage with each Generative AI, compare the quality of the code, and contrast their features and limitations. An ITAM system is seemingly straightforward; you manage assets, tagging and tracking. As we will see, however, there is considerable complexity that is hidden within the requirements. This makes this project an excellent candidate for using a Generative AI to survey the problem-space. Let’s start with an introduction to what we are trying to build.

2.1 The Information Technology Asset Management System

An *Information Technology Asset Management* (ITAM) system is a tool to manage and track hardware devices, software licenses, and other IT-related components throughout their lifecycle. ITAM systems typically consist of hardware and software inventory tools, license management software, and other related software applications. The system may also involve manual processes and physical tracking of IT assets using QR codes, barcodes, or other physical asset management technologies.

Generally, ITAM systems will have a centralized database, which stores the asset identifiers and attributes specific to the asset type. For example, you might store device type, model number, operating system, and installed applications for desktop PCs. For software, you might hold the application's name, the vendor, the number of licenses available, and on which computers the software has been installed. The latter ensures that your organization complies with all licensing restrictions. By monitoring usage, you should never exceed the number of licenses you have purchased.

ITAM systems also confer the ability to control costs. Since you always know what software and hardware you have available, you should not have to make any unnecessary purchases. These systems centralized purchases, which can help with volume purchasing. Unused hardware can be sold. Underutilized hardware can have their workloads consolidated. Additionally, as we will see, we can use this purchase date information to calculate the depreciation value of the hardware, applying this value against your organization's taxes.

We will explore more of the features of an ITAM system, building a miniature version, one focused on hardware management. We will begin with ChatGPT, building out the first component using Python. We will then replicate this exercise using Copilot and AWS CodeWhisperer. By repeating this process with each of these Generative AI tools, we can contrast the differences and correlate the similarities.

First, starting with ChatGPT, we will focus on building out the Asset class. Next, we will develop a means to present it via a representational state transfer (REST) controller using the library FastAPI. Finally, we will build on this in subsequent chapters, adding features, extending the design, and adding documentation.

Why Python?

In 2023, one should not need to justify the decision to use Python 3 in one's application development. However, it is worth enumerating why choosing Python for this book makes sense, given the topic. Firstly, Python is an excellent language for prototyping projects. Data scientists and machine learning specialists use it frequently for this reason. Secondly, it is expressive and productive. A few lines can get a good deal done, especially given the size of the standard library, not to mention the commonly used libraries such as pandas and numpy. Thirdly, it is easy to learn and use. Its syntax focuses on (and enforces) readability. Finally, it is supported by all of the Generative AI platforms.

2.2 A Foray into ChatGPT

Context is one of the most important aspects of working with ChatGPT. Your previous *prompts* can drastically change the results from your current prompt. In the context of language models like ChatGPT, a prompt refers to the input provided to the model to generate a response. it can be a single sentence, a paragraph, or even a longer piece of text. It serves as the instruction or query to the model, guiding its response. Given the quality of the prompt forms the context in which the model responds, it is important to always be aware of what prompts you have issued in the current session. Therefore, starting with a new session every time you begin a new project is advised. Appendix A will walk you through setting up an account, logging into ChatGPT, and writing your first prompt.

We will start with the simple prompt, "Using Python, please write an Asset class for an ITAM." You should receive the following output or something very similar.

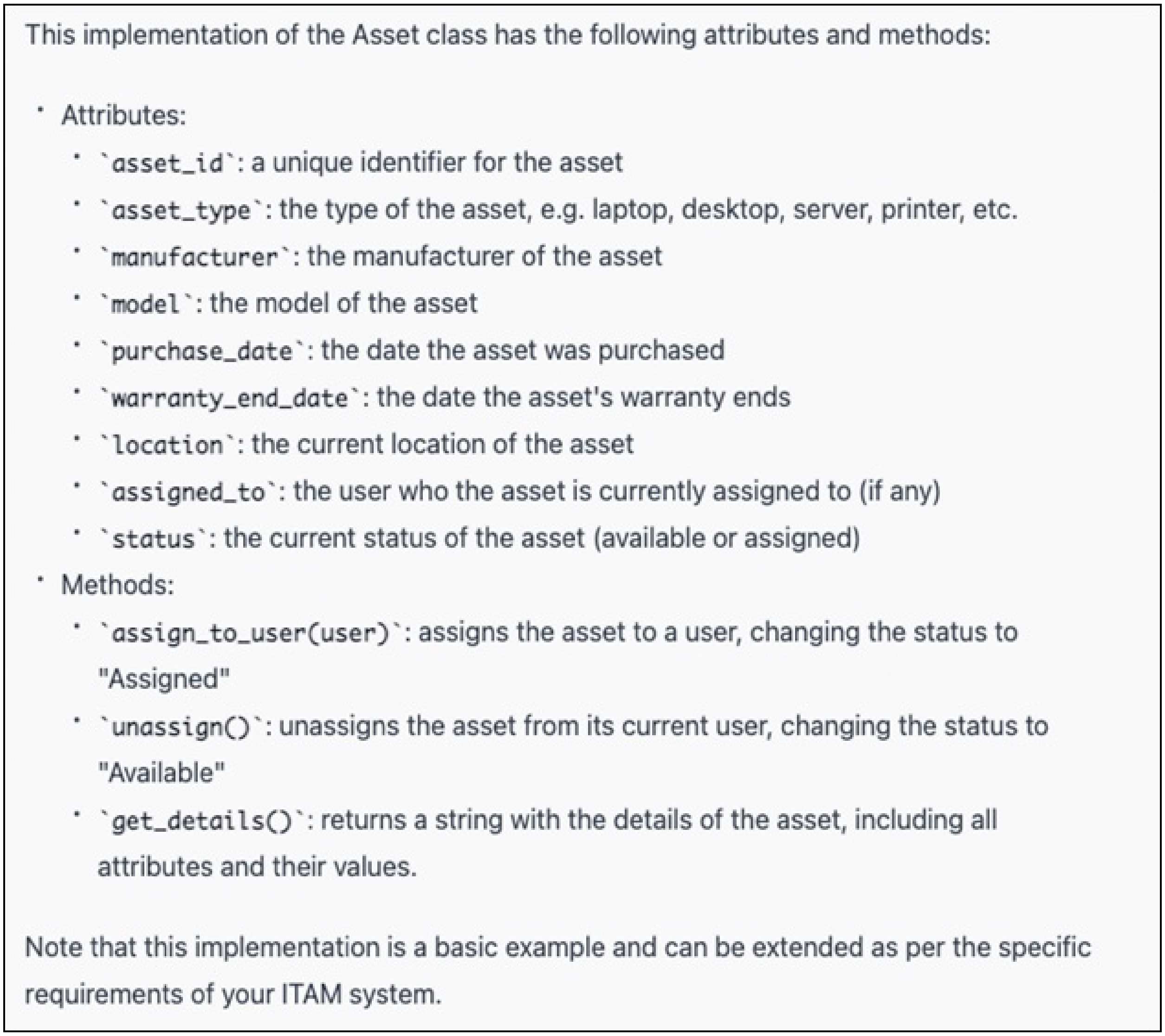
Listing 2.1 The Asset class generated by ChatGPT when asked to create a class for an ITAM

```
class Asset:
def __init__(self, asset_id, asset_type, manufacturer, model, pur
self.asset_id = asset_id
self.asset_type = asset_type
self.manufacturer = manufacturer
self.model = model
self.purchase_date = purchase_date
self.warranty_end_date = warranty_end_date
self.location = location
self.assigned_to = None
self.status = 'Available'
```

```
def assign_to_user(self, user):
self.assigned_to = user
self.status = 'Assigned'
def unassign(self):
self.assigned_to = None
self.status = 'Available'
def get_details(self):
details = f "Asset ID: {self.asset_id}\n"
details += f"Type: {self.asset_type}\n"
details += f"Manufacturer: {self.manufacturer}\n"
details += f"Model: {self.model}\n"
details += f "Purchase Date: {self.purchase_date}\n"
details += f "Warranty End Date: {self.warranty_end_date}\n"
details += f"Location: {self.location}\n"
details += f "Assigned To: {self.assigned_to}\n"
details += f"Status: {self.status}\n"
return details
```

In addition to the code, ChatGPT will return documentation for this class.

Figure 2.1 ChatGPT enumerates the attributes in the Asset class as well as the class's methods



This implementation of the Asset class has the following attributes and methods:

- Attributes:
  - `asset_id`: a unique identifier for the asset
  - `asset_type`: the type of the asset, e.g. laptop, desktop, server, printer, etc.
  - `manufacturer`: the manufacturer of the asset
  - `model`: the model of the asset
  - `purchase_date`: the date the asset was purchased
  - `warranty_end_date`: the date the asset's warranty ends
  - `location`: the current location of the asset
  - `assigned_to`: the user who the asset is currently assigned to (if any)
  - `status`: the current status of the asset (available or assigned)
- Methods:
  - `assign_to_user(user)`: assigns the asset to a user, changing the status to "Assigned"
  - `unassign()`: unassigns the asset from its current user, changing the status to "Available"
  - `get_details()`: returns a string with the details of the asset, including all attributes and their values.

Note that this implementation is a basic example and can be extended as per the specific requirements of your ITAM system.

To track our Assets, we will need a way to interact with them. An API is an excellent way to interact with objects in your system. REST is one way to expose your API functionality in a microservice architecture. Therefore, we will ask ChatGPT to create a REST controller for this class with the following prompt, "Please create a REST controller for this class." The abridged ChatGPT generated code is in listing 2.2.

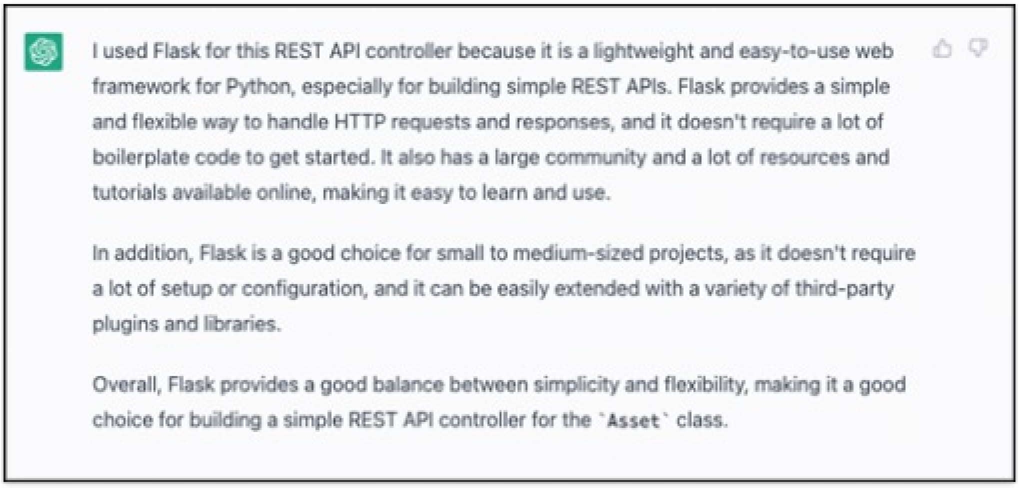
Listing 2.2 The REST controller generated by ChatGPT when asked to add a REST controller to the project

```
from flask import Flask, request, jsonify
from asset import Asset, AssetManager
app = Flask(__name__)
```

```
asset_manager = AssetManager()
# Get all assets
@app.route('/assets', methods=['GET'])
def get_assets():
assets = [asset.__dict__ for asset in asset_manager.assets]
return jsonify({'assets': assets})
# Get an asset by ID
@app.route('/assets/<string:asset_id>', methods=['GET'])
def get_asset(asset_id):
asset = asset_manager.get_asset_by_id(asset_id)
if asset:
return jsonify(asset.__dict__)
else:
return jsonify({'error': 'Asset not found'})
if __name__ == '__main__':
app.run(debug=True)
```

ChatGPT made some interesting assumptions. Firstly, it used Flask to build out the REST controller. While Flask is a popular, Python based web framework and by no means a bad decision, it would be of interest to us why it was selected. So, we can ask ChatGPT why it chose Flask. Use the following prompt, "Why did you use Flask for this controller?"

Figure 2.2 ChatGPT explains the rationale for using Flask when it was not provided a framework to use. It states that it is lightweight and easy to use.



I used Flask for this REST API controller because it is a lightweight and easy-to-use web framework for Python, especially for building simple REST APIs. Flask provides a simple and flexible way to handle HTTP requests and responses, and it doesn't require a lot of boilerplate code to get started. It also has a large community and a lot of resources and tutorials available online, making it easy to learn and use.

In addition, Flask is a good choice for small to medium-sized projects, as it doesn't require a lot of setup or configuration, and it can be easily extended with a variety of third-party plugins and libraries.

Overall, Flask provides a good balance between simplicity and flexibility, making it a good choice for building a simple REST API controller for the `Asset` class.

While the rationale is good, it is not the framework we wanted to use. Shortly, we will have ChatGPT switch the application to use FastAPI. First, however, we should ensure that the code we just “wrote” will run. However, we are missing some dependencies in our project. We can ask ChatGPT to create a requirements.txt file for us.

Enter the following prompt in the input field, "Please create a requirements.txt file for this project." ChatGPT should respond with the following code snippet.

Listing 2.3 The requiremnts.txt file for the Asset management project with Flask support

```
click==8.0.3
Flask==2.1.1
importlib-metadata==4.8.1
itsdangerous==2.0.1
Jinja2==3.0.2
MarkupSafe==2.0.1
typing-extensions==3.10.0.2
Werkzeug==2.0.1
zipp==3.6.0
```

Once we create and populate the initial app.py file (the initial Python code that ChatGPT created) and the requirements.txt file we just generated, we will notice the second assumption that ChatGPT made: the AssetManager class. ChatGPT silently generated that code without cluing us in. This hidden class for which we have no source code means our project will not run.

Thankfully, we can ask ChatGPT to give us this code. Enter the following prompt, "Please show me the source code to AssetManager."

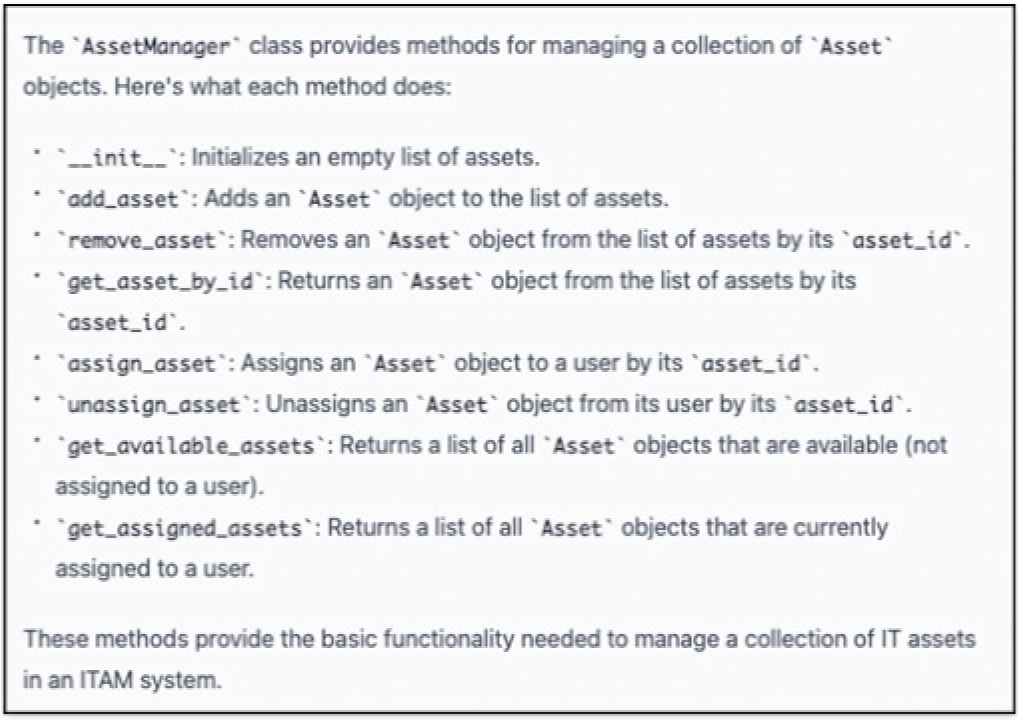
Listing 2.4 The class definition for the AssetManager class generated by ChatGPT

```
class AssetManager:
def __init__(self):
self.assets = []
def add_asset(self, asset):
self.assets.append(asset)
def remove_asset(self, asset_id):
for asset in self.assets:
if asset.asset_id == asset_id:
self.assets.remove(asset)
return True
return False
def get_asset_by_id(self, asset_id):
```

```
for asset in self.assets:
if asset.asset_id == asset_id:
return asset
return None
def assign_asset(self, asset_id, user):
asset = self.get_asset_by_id(asset_id)
if asset and asset.status == 'Available':
asset.assign_to_user(user)
return True
return False
def unassign_asset(self, asset_id):
asset = self.get_asset_by_id(asset_id)
if asset and asset.status == 'Assigned':
asset.unassign()
return True
return False
def get_available_assets(self):
return [asset for asset in self.assets if asset.status == 'Availa
def get_assigned_assets(self):
return [asset for asset in self.assets if asset.status == 'Assign
```

Additionally, ChatGPT will produce documentation listing the class usage, as well as all methods of the class. This documentation would be helpful if we were looking how to use the class, or if we wanted to add comments to our code.

Figure 2.3 ChatGPT provides documentation on the AssetManager class, including the attributes and methods.



The `AssetManager` class provides methods for managing a collection of `Asset` objects. Here's what each method does:

- `__init__`: Initializes an empty list of assets.
- `add_asset`: Adds an `Asset` object to the list of assets.
- `remove_asset`: Removes an `Asset` object from the list of assets by its `asset_id`.
- `get_asset_by_id`: Returns an `Asset` object from the list of assets by its `asset_id`.
- `assign_asset`: Assigns an `Asset` object to a user by its `asset_id`.
- `unassign_asset`: Unassigns an `Asset` object from its user by its `asset_id`.
- `get_available_assets`: Returns a list of all `Asset` objects that are available (not assigned to a user).
- `get_assigned_assets`: Returns a list of all `Asset` objects that are currently assigned to a user.

These methods provide the basic functionality needed to manage a collection of IT assets in an ITAM system.

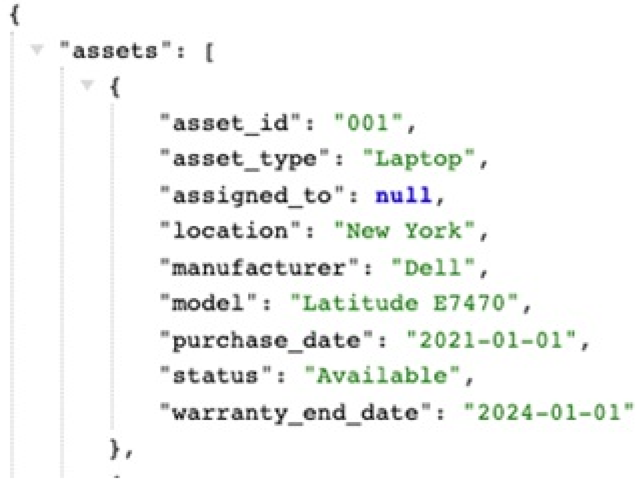
Should you choose to store the class AssetManager in a separate Python file, which is a good idea, then you will need to update the app.py file, changing the import statements this way:

Listing 2.5 Fixing the import statements of the file app.py

```
from flask import Flask, request, jsonify
from asset import Asset
from asset_manager import AssetManager
app = Flask(__name__)
asset_manager = AssetManager()
…
```

You should be able to run our very simple ITAM system now. Once you start the Flask application, If you navigate to http://localhost:5000/assets, you should see Assets as JSON output.

Figure 2.4 We added three Assets to the AssetManager. The controller returned this collection in JSON format



```
{
  "assets": [
    {
      "asset_id": "001",
      "asset_type": "Laptop",
      "assigned_to": null,
      "location": "New York",
      "manufacturer": "Dell",
      "model": "Latitude E7470",
      "purchase_date": "2021-01-01",
      "status": "Available",
      "warranty_end_date": "2024-01-01"
    },
```

There is one outstanding issue to which we should turn out attention. This project is still using Flask. There are several reasons why we prefer to use FastAPI instead of Flask. The first of which is that FastAPI would give us better performance. FastAPI consistently ranks significantly better in benchmarks, thus the name. FastAPI has support for asynchronous non-blocking, highly concurrent programming. Asynchronous non-blocking systems yield better performance on the same hardware because the computer will execute other tasks while it is waiting for I/O operations to complete.

Therefore, we should switch to FastAPI. We can ask ChatGPT to do this for us. Enter the following prompt, "Please change this project from using Flask to using FastAPI." ChatGPT will generate code similar to the following app.py. Note that listing 2.6 is abridged, including only the GET REST operations. The full output would have support for POST, PUT, and DELETE.

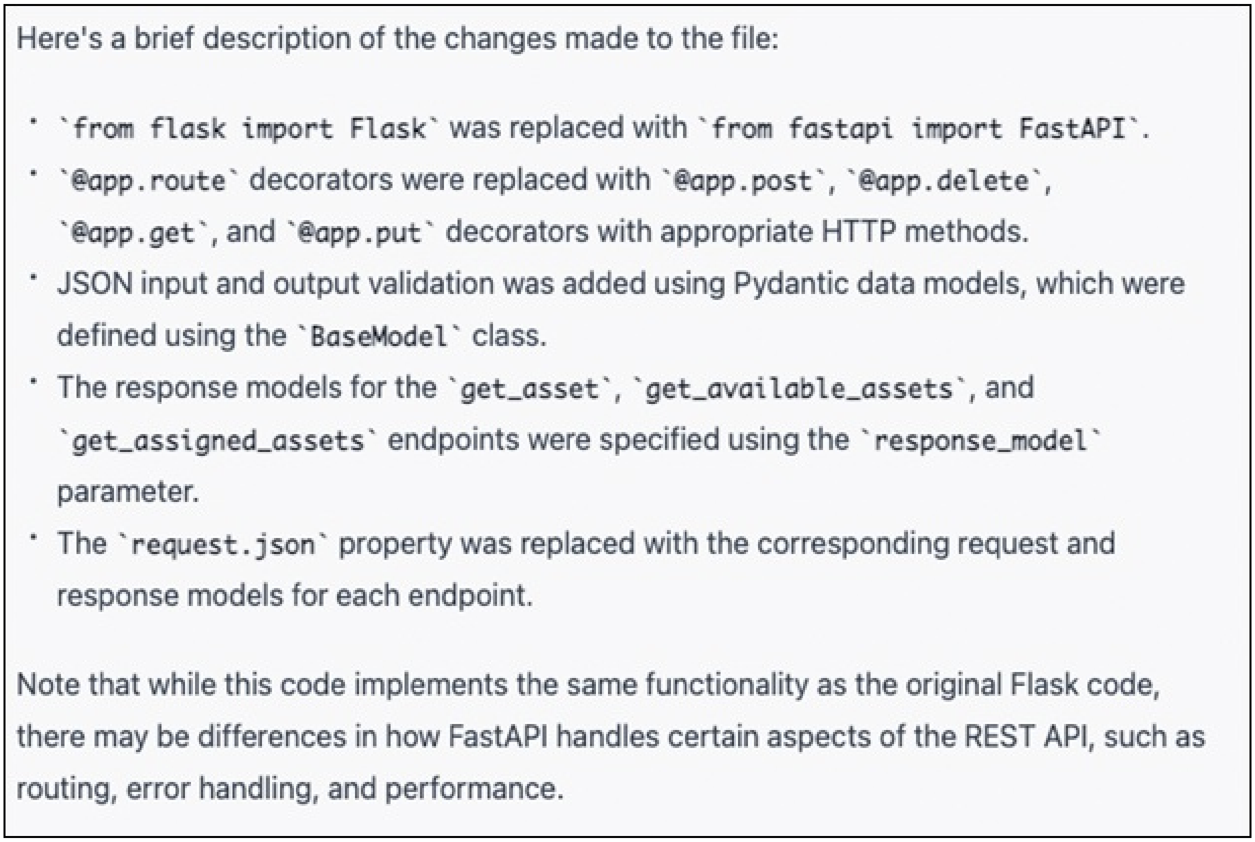
Listing 2.6 The updated app.py which includes FastAPI support

```
app = FastAPI()
asset_manager = AssetManager()
class AssetIn(BaseModel):
asset_id: str
asset_type: str
manufacturer: str
model: str
purchase_date: str
warranty_end_date: str
location: str
```

```
class AssetOut(BaseModel):
asset_id: str
asset_type: str
manufacturer: str
model: str
purchase_date: str
warranty_end_date: str
location: str
assigned_to: str
status: str
@app.get("/assets/{asset_id}", response_model=AssetOut)
async def get_asset(asset_id: str):
asset = asset_manager.get_asset_by_id(asset_id)
if asset:
return asset
else:
return {"message": "Asset not found"}
```

There are several things to note. The first is that ChatGPT informs us of the changes it made to the application. See figure 2.5. Next, notice that it created two additionally embedded classes: AssetIn and AssetOut. This is considered good practice, as it provides an interface or contract the clients should adhere to, while allowing us the freedom to change our internal representation of our domain model.

Figure 2.5 ChatGPT explains the changes that it made to the application to change it from a Flask application to FastAPI.


Here's a brief description of the changes made to the file:

- `from flask import Flask` was replaced with `from fastapi import FastAPI`.
- `@app.route` decorators were replaced with `@app.post`, `@app.delete`, `@app.get`, and `@app.put` decorators with appropriate HTTP methods.
- JSON input and output validation was added using Pydantic data models, which were defined using the `BaseModel` class.
- The response models for the `get_asset`, `get_available_assets`, and `get_assigned_assets` endpoints were specified using the `response_model` parameter.
- The `request.json` property was replaced with the corresponding request and response models for each endpoint.

Note that while this code implements the same functionality as the original Flask code, there may be differences in how FastAPI handles certain aspects of the REST API, such as routing, error handling, and performance.

When changing the middleware of a project, you should check to see if the domain model classes have not changed. In this case, both Asset and AssetManager have changed. We should ask ChatGPT to show us those new classes.

Note

If you have middleware, you should ensure that your domain model has not be affected.

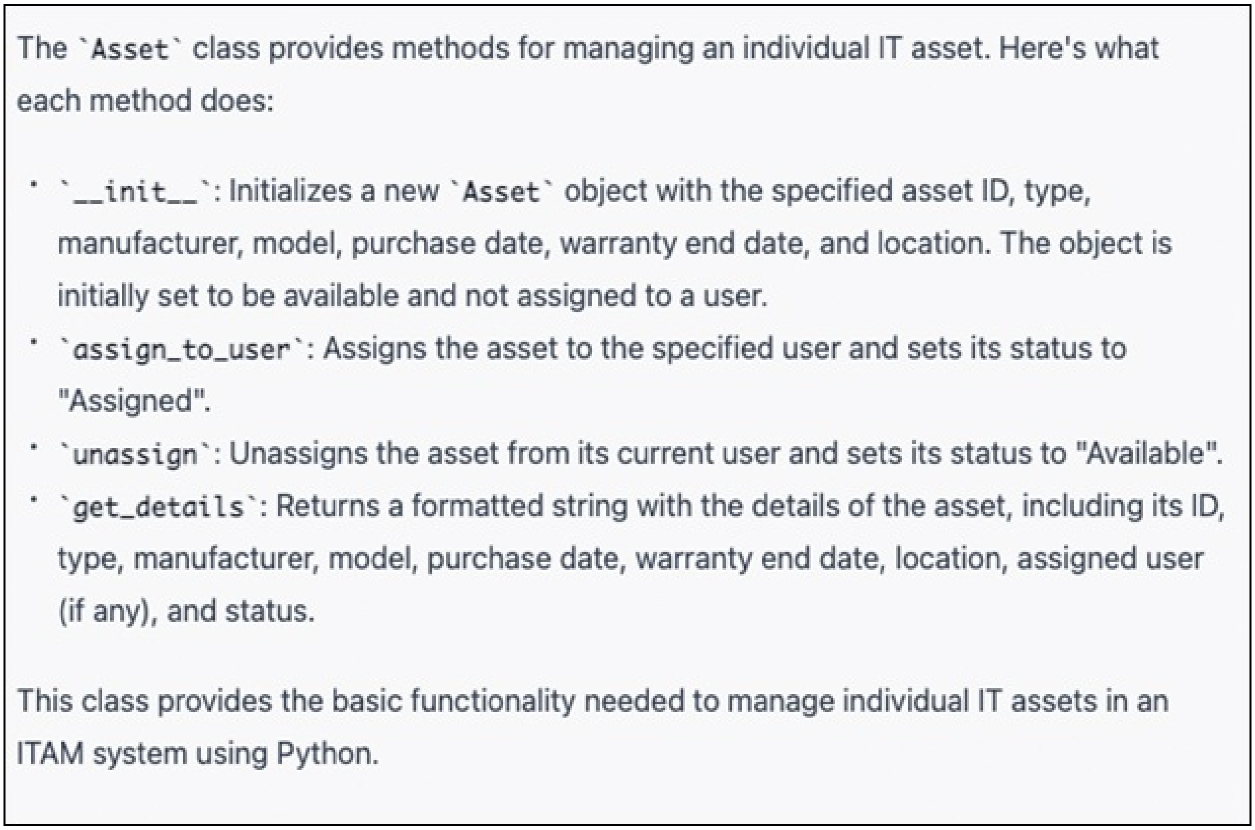
Enter the following prompt: "Please show the code for Asset."

Listing 2.7 The updated Asset class reflecting the change in middleware from Flask to FastAPI

```
from typing import Optional
class Asset:
def __init__(self, asset_id: str, asset_type: str, manufacturer:
self.asset_id = asset_id
self.asset_type = asset_type
self.manufacturer = manufacturer
self.model = model
self.purchase_date = purchase_date
self.warranty_end_date = warranty_end_date
self.location = location
self.assigned_to: Optional[str] = None
self.status: str = 'Available'
def assign_to_user(self, user: str):
self.assigned_to = user
self.status = 'Assigned'
def unassign(self):
self.assigned_to = None
self.status = 'Available'
def get_details(self):
details = f "Asset ID: {self.asset_id}\n"
details += f"Type: {self.asset_type}\n"
details += f"Manufacturer: {self.manufacturer}\n"
details += f"Model: {self.model}\n"
details += f "Purchase Date: {self.purchase_date}\n"
details += f "Warranty End Date: {self.warranty_end_date}\n"
details += f"Location: {self.location}\n"
details += f "Assigned To: {self.assigned_to}\n"
details += f"Status: {self.status}\n"
return details
```

The most meaningful change that ChatGPT added the Optional type hint. This would signal your intent that this attribute – assigned_to – should not be Nil and therefore cause a NoneType error. As usual, ChatGPT also helpfully included method documentation.

Figure 2.6 ChatGPT explains the updated Asset class. While it does not expressly tell us the changes that it enacted when the class was updates, it does provide fulsome documentation on the class's methods.



The `Asset` class provides methods for managing an individual IT asset. Here's what each method does:

- `__init__`: Initializes a new `Asset` object with the specified asset ID, type, manufacturer, model, purchase date, warranty end date, and location. The object is initially set to be available and not assigned to a user.
- `assign_to_user`: Assigns the asset to the specified user and sets its status to "Assigned".
- `unassign`: Unassigns the asset from its current user and sets its status to "Available".
- `get_details`: Returns a formatted string with the details of the asset, including its ID, type, manufacturer, model, purchase date, warranty end date, location, assigned user (if any), and status.

This class provides the basic functionality needed to manage individual IT assets in an ITAM system using Python.

Next, enter the following prompt: `"Please show the code for AssetManager."` The constructor initializes a List of Assets. This List is the data structure on which the AssetManager class will use to perform the CRUD (Create, Read, Update, and Delete) operations.

Listing 2.8 The updated AssetManager class reflecting the change in middleware from Flask to FastAPI

```
from typing import List, Optional
from asset import Asset
class AssetManager:
def __init__(self):
self.assets: List[Asset] = []
def add_asset(self, asset: Asset):
self.assets.append(asset)
```

The remove_asset method will iterate over the elements in the List of Assets. If it finds the Asset with the identifier supplied in the method's parameter (asset_id), it will remove it from the List.

Listing 2.9 The remove_asset method of the AssetManager

```
def remove_asset(self, asset_id: str) -> bool:
for asset in self.assets:
if asset.asset_id == asset_id:
self.assets.remove(asset)
return True
return False
```

The `get_asset` method iterates over all elements in the List, looking for the Asset with the asset_id passed in as a parameter. If it finds this Asset then it will return it. This search functionality is very similar to many of the methods in this class: you will notice this again in the `assign_asset` and `unassign_asset` methods. If we were to observe the DRY (Don't Repeat Yourself) principle (and we should!), then we would refactor this class (or ask ChatGPT to do it for us), consolidating the logic by exacting a single method for use throughout the class.

Listing 2.10 The common methods of the AssetManager class

```
def get_asset_by_id(self, asset_id: str) -> Optional[Asset]:
for asset in self.assets:
if asset.asset_id == asset_id:
return asset
return None
def assign_asset(self, asset_id: str, user: str) -> bool:
for asset in self.assets:
if asset.asset_id == asset_id:
if asset.status == 'Available':
asset.assign_to_user(user)
return True
else:
return False
return False
def unassign_asset(self, asset_id: str) -> bool:
for asset in self.assets:
if asset.asset_id == asset_id:
if asset.status == 'Assigned':
asset.unassign()
return True
else:
return False
return False
```

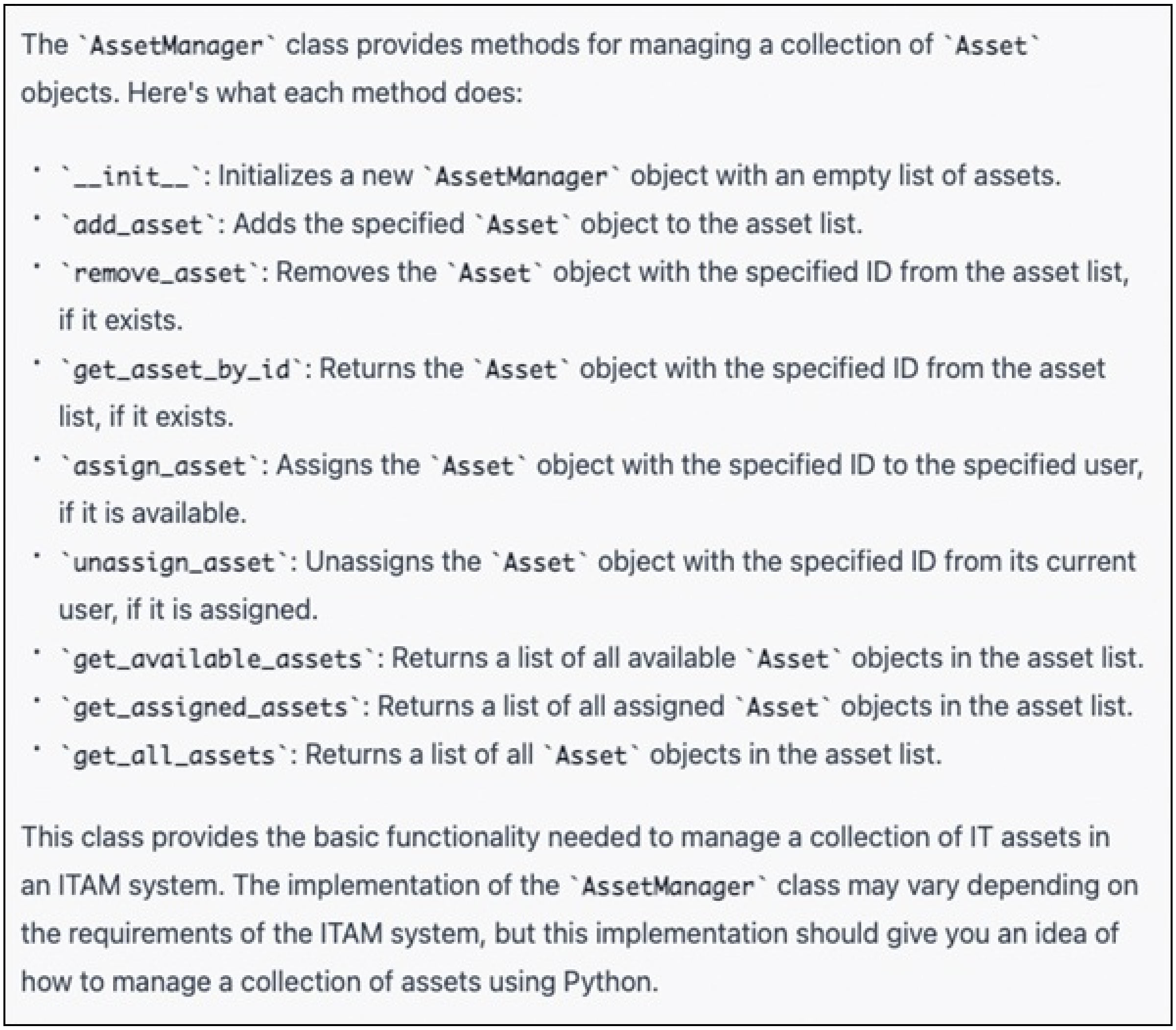
Similarly, the get_available_asset and get_assigned_asset methods contain common functionality. Both find all Assets whose status matches the parameter. These would be candidates for refactoring the code into a common methods that each uses.

Listing 2.11 Methods to return Assets with a given status

```
def get_available_assets(self) -> List[Asset]:
return [asset for asset in self.assets if asset.status == 'Availa
def get_assigned_assets(self) -> List[Asset]:
return [asset for asset in self.assets if asset.status == 'Assign
def get_all_assets(self) -> List[Asset]:
return self.assets
```

Accompanying this code was documentation related to the class's methods and their usage.

Figure 2.7 ChatGPT lists the methods that are available in the AssetManager class, now that it has been migrated to use FastAPI



The `AssetManager` class provides methods for managing a collection of `Asset` objects. Here's what each method does:

- `__init__`: Initializes a new `AssetManager` object with an empty list of assets.
- `add_asset`: Adds the specified `Asset` object to the asset list.
- `remove_asset`: Removes the `Asset` object with the specified ID from the asset list, if it exists.
- `get_asset_by_id`: Returns the `Asset` object with the specified ID from the asset list, if it exists.
- `assign_asset`: Assigns the `Asset` object with the specified ID to the specified user, if it is available.
- `unassign_asset`: Unassigns the `Asset` object with the specified ID from its current user, if it is assigned.
- `get_available_assets`: Returns a list of all available `Asset` objects in the asset list.
- `get_assigned_assets`: Returns a list of all assigned `Asset` objects in the asset list.
- `get_all_assets`: Returns a list of all `Asset` objects in the asset list.

This class provides the basic functionality needed to manage a collection of IT assets in an ITAM system. The implementation of the `AssetManager` class may vary depending on the requirements of the ITAM system, but this implementation should give you an idea of how to manage a collection of assets using Python.

Finally, we will need ChatGPT to regenerate the dependencies file as well. Use the following prompt, `"Please show me the new requirements.txt file."`

Listing 2.12 The updated requirements.txt file, which uses FastAPI

```
fastapi==0.70.0
uvicorn==0.15.0
pydantic==1.8.2
```

Once you have re-run the `pip` command (`pip install -r requirements.txt`) to install the updated dependencies, you should be able to run the project using the following command: `uvicorn app:app –-reload`.

Note

the port has now changed to 8000. When I ran this, there wasn't a method to display all Assets in the AssetManager. We can ask ChatGPT to add such a method. "Please add a method to app.py to return and display all Assets."

Listing 2.13 The method used to display all of the Assets in the AssetManager

```
@app.get("/assets/", response_model=List[AssetOut])
async def get_assets():
assets = asset_manager.get_all_assets()
return assets
```

Routing in FastAPI

In this method, we define a new route using the @app.get decorator, with a URL path of /assets/. We also specify a response model of List[AssetOut] using the response_model parameter, which tells FastAPI to validate the output data and generate API documentation based on the schema of the AssetOut data model.

Inside the method, we call the get_all_assets method of the asset_manager object, which returns a list of all Asset objects. We then return this list to the client, which will be automatically converted to a JSON array by FastAPI.

You can test this endpoint using a web browser or a tool such as curl. For example, if you are running the application locally on port 8000, you could open http://localhost:8000/assets/ in a web browser to see a JSON array of all assets.

Let's test our ITAM system by adding and displaying a new Asset in our browser. You can use curl or Postman to add a new Asset.

Listing 2.14 Using curl to add new Asset to the AssetManager

```
curl -X POST -H "Content-Type: application/json" -d '{"asset_id":
```

You will likely get an error when you attempt to buy this new asset. Two modifications are required to get the code that ChatGPT generated to work.

The first is to add a new method to the Asset class called `to_dict()`. FastAPI uses the library pydantic to convert the In/Out models (e.g. `AssetIn` and `AssetOut`) to and from the Asset class. It does this by using a dictionary to assign the attributes. The `to_dict()` method will give us an easy way to provide that dictionary.

Listing 2.15 Add the to_dict method to the Asset class defined in asset.py

```
def to_dict(self):
return {
"asset_id": self.asset_id,
"asset_type": self.asset_type,
"manufacturer": self.manufacturer,
"model": self.model,
"purchase_date": self.purchase_date,
"warranty_end_date": self.warranty_end_date,
"location": self.location,
"assigned_to": self.assigned_to,
"status": self.status
}
```

The second change we need to do is to translate the output model (`AssetOut`) to JSON. We would do that everywhere that we output this object. We will examine the changes in the class that relate to changing the output to JSON.

Listing 2.16 An updated app.py class with JSON encoding of output

```
@app.get("/assets/", response_model=List[AssetOut])
async def get_assets():
assets = asset_manager.get_all_assets()
return JSONResponse(content=[asset.to_dict() for asset in assets]
@app.post("/assets/")
async def add_asset(asset: AssetIn):
new_asset = Asset(asset.asset_id, asset.asset_type, asset.manufac
asset_manager.add_asset(new_asset)
return {"message": "Asset added successfully"}
@app.delete("/assets/{asset_id}")
async def remove_asset(asset_id: str):
if asset_manager.remove_asset(asset_id):
return {"message": "Asset removed successfully"}
else:
return {"message": "Asset not found"}
@app.get("/assets/{asset_id}", response_model=AssetOut)
async def get_asset(asset_id: str):
```

```
asset = asset_manager.get_asset_by_id(asset_id)
if asset:
return JSONResponse(content=asset.to_dict())
else:
return {"message": "Asset not found"}
@app.put("/assets/{asset_id}/assign")
async def assign_asset(asset_id: str, user: str):
if asset_manager.assign_asset(asset_id, user):
return {"message": "Asset assigned successfully"}
else:
return {"message": "Asset not available"}
@app.put("/assets/{asset_id}/unassign")
async def unassign_asset(asset_id: str):
if asset_manager.unassign_asset(asset_id):
return {"message": "Asset unassigned successfully"}
else:
return {"message": "Asset not assigned"}
@app.get("/assets/available/", response_model=List[AssetOut])
async def get_available_assets():
assets = asset_manager.get_available_assets()
return JSONResponse(content=[asset.to_dict() for asset in assets]
@app.get("/assets/assigned/", response_model=List[AssetOut])
async def get_assigned_assets():
assets = asset_manager.get_assigned_assets()
return JSONResponse(content=[asset.to_dict() for asset in assets]
```

Switching the middleware from Flask to FastAPI caused many issues in our code. You should try creating a prompt to create this project using FastAPI first and compare that against the project that we created iteratively. You should notice that the code is drastically different. I leave that as an exercise for the reader.

2.3 Let Copilot take control!

Let us now use GitHub Copilot to tackle the creation of the Asset class. Appendix B has instructions on how to create an account and install the plugin into your favorite IDE (assuming your favorite IDE is either VS Code or PyCharm). Once you have completed the installation you should create a new project in your favorite IDE. First, create a new file named asset.py. This class will hold the source code for the Asset class. As you type the constructor, Copilot will make code suggestions. You will likely get code resembling the source code in figure 2.8. Alternatively, you could activate interactive model by pressing `Ctrl+Enter`, and Copilot will generate up to ten suggestions based off of the current context.

NOTE

Throughout the book we will mostly use inline mode, in which you use the Tab key to accept a given suggestion.

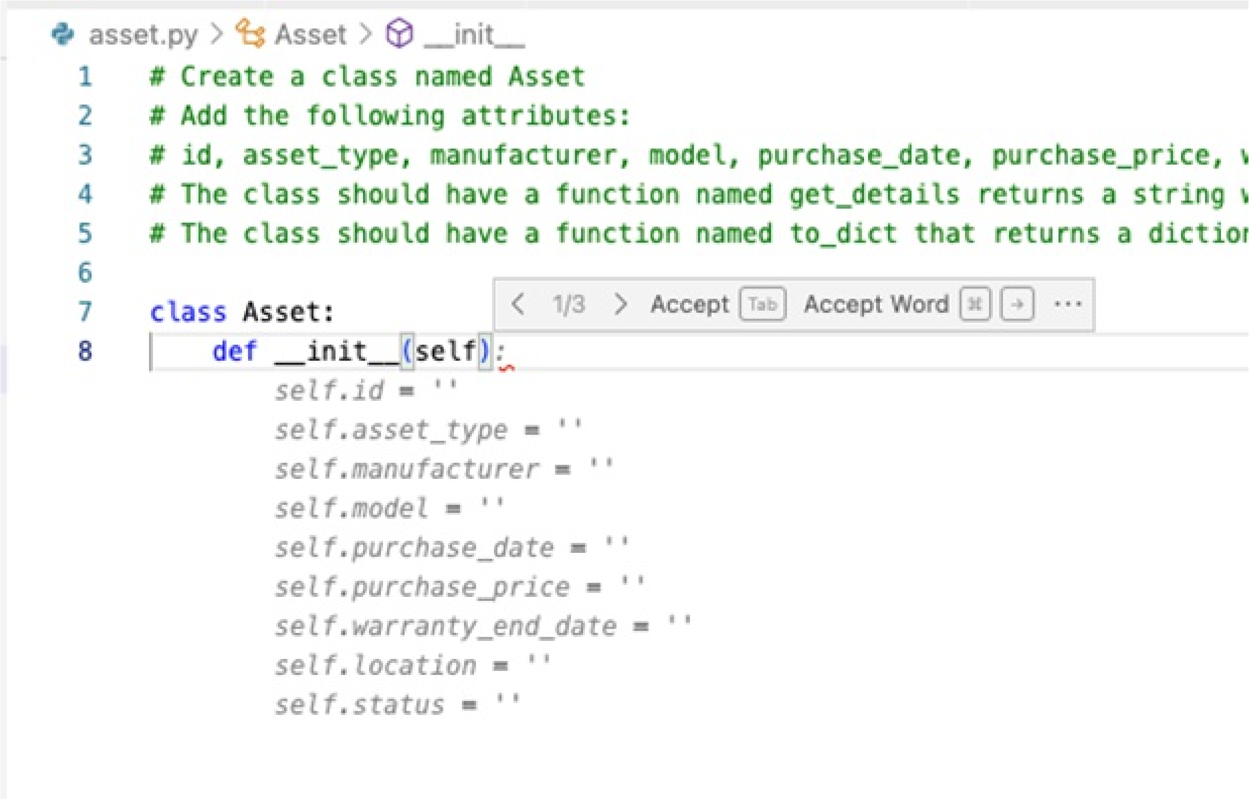
While Copilot's first attempt was admirable, it does not resemble the class definition we were interested in; we want an ITAM system asset, not a generic implementation of an asset. Like what we did with ChatGPT, we will create a prompt that will signal our intention to Copilot, giving it additional context and information that it can use to generate a code suggestion. Entering prompts in Copilot is as simple as adding a comment at the beginning of the source code file. We will tell Copilot what the name of the class should be and what attributes it should have.

Listing 2.17 The starting prompt for GitHub Copilot asks it to create a new class named Asset

```
# Create a class named Asset
# Add the following attributes: id, asset_type, manufacturer, mod
# The class should have a function named get_details that returns
# The class should have a function named to_dict that returns a d
```

As you begin to define the constructor, Copilot will return an entire code suggestion for an ITAM system asset. If we made a side-by-side comparison of the ChatGPT-generated code with the resulting code from Copilot, we would notice that they closely resemble one another.

Figure 2.8 Based off of the comment that we added to top of the file, Copilot was able to generate a constructor, containing all of the attributes included in the prompt.



```
# Create a class named Asset
# Add the following attributes:
# id, asset_type, manufacturer, model, purchase_date, purchase_price,
# The class should have a function named get_details returns a string
# The class should have a function named to_dict that returns a dictio

class Asset:
    def __init__(self):
        self.id = ''
        self.asset_type = ''
        self.manufacturer = ''
        self.model = ''
        self.purchase_date = ''
        self.purchase_price = ''
        self.warranty_end_date = ''
        self.location = ''
        self.status = ''
```

Next, add the methods `get_details` and `to_dict`. The JSONResponse class accepts a dictionary and convert it to JSON for us. The `to_dict` method will help facilitate this handoff. The `get_details` method would be useful during debugging, enabling us to see the contents of the class.

Listing 2.18 The code that Copilot generates in response to our request

```
# Create a class named Asset
# Add the following attributes: id, asset_type, manufacturer, mod
# The class should have a function named get_details that returns
# The class should have a function named to_dict that returns a d
class Asset:
def __init__(self):
self.id = None
self.asset_type = None
self.manufacturer = None
self.model = None
self.purchase_date = None
self.purchase_price = None
self.warranty_end_date = None
self.location = None
self.status = None
```

```
def get_details(self):
return f"{self.id}, {self.asset_type}, {self.manufacturer}, {self
def to_dict(self):
return {
"id": self.id,
"asset_type": self.asset_type,
"manufacturer": self.manufacturer,
"model": self.model,
"purchase_date": self.purchase_date,
"purchase_price": self.purchase_price,
"warranty_end_date": self.warranty_end_date,
"location": self.location,
"status": self.status
}
```

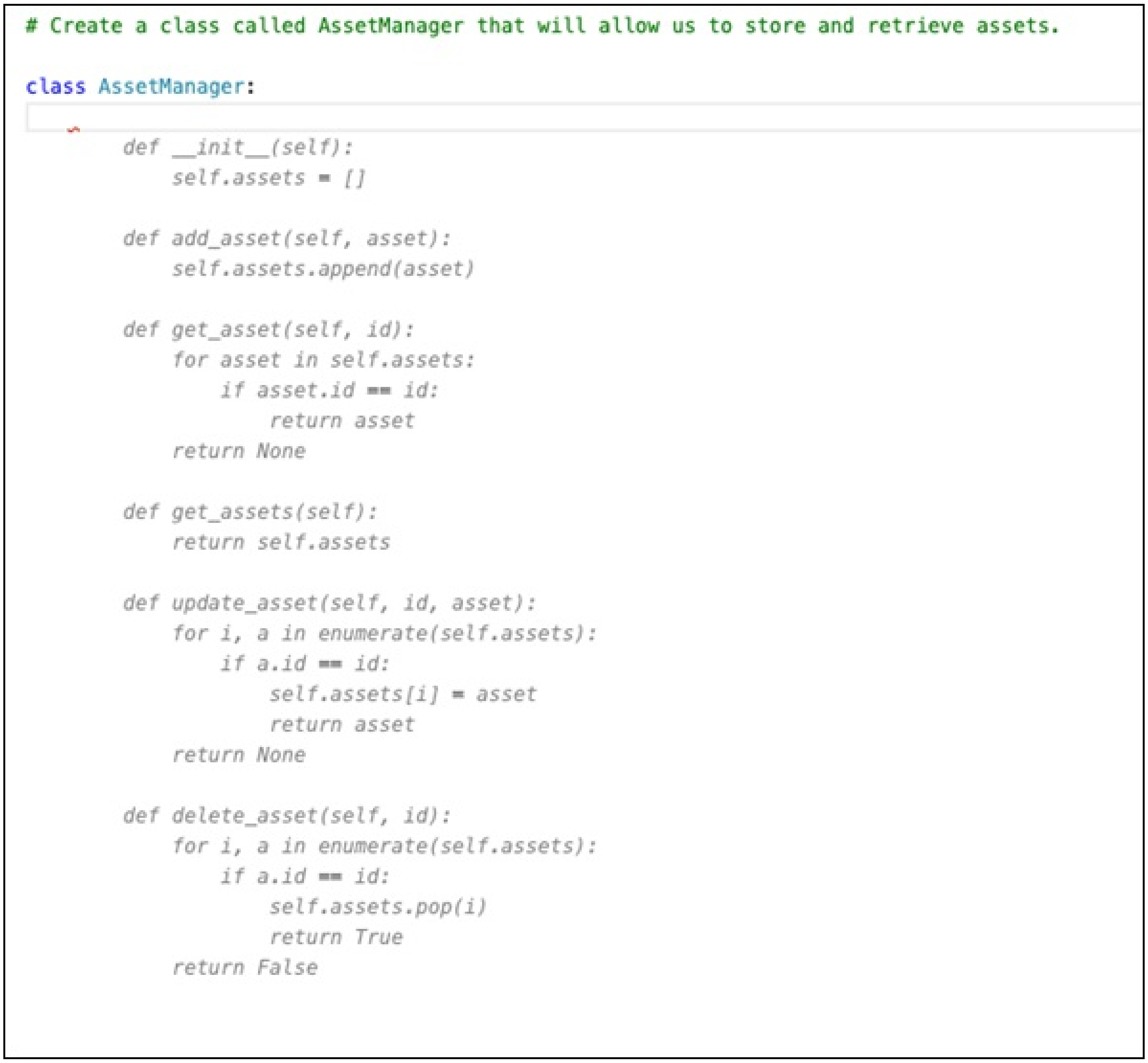
Now that we have the asset class. We will move on to the AssetManager class. We will start with a basic prompt, telling Copilot to create the AssetManager class and define some basic functionality.

Listing 2.19 The prompt to have Copilot generate our AssetManager

```
# Create a class called AssetManager that will allow us to store
```

Shortly after defining the class, you will notice that Copilot will suggest the complete source code for this manager.

Figure 2.9 With minimal instruction Copilot was able to create an AssetManager that allowed for the storage and retrieval of Assets.



```
# Create a class called AssetManager that will allow us to store and retrieve assets.

class AssetManager:

    def __init__(self):
        self.assets = []

    def add_asset(self, asset):
        self.assets.append(asset)

    def get_asset(self, id):
        for asset in self.assets:
            if asset.id == id:
                return asset
        return None

    def get_assets(self):
        return self.assets

    def update_asset(self, id, asset):
        for i, a in enumerate(self.assets):
            if a.id == id:
                self.assets[i] = asset
                return asset
        return None

    def delete_asset(self, id):
        for i, a in enumerate(self.assets):
            if a.id == id:
                self.assets.pop(i)
                return True
        return False
```

Copilot's source code for the AssetManager will closely resemble the code that ChatGPT generated.

Listing 2.20 The complete AssetManager source code generated by Copilot from our initial prompt

```
# Create a class called AssetManager that will allow us to store
class AssetManager:
def __init__(self):
self.assets = []
def add_asset(self, asset):
self. Assets.append(asset)
def get_assets(self):
return self. assets
def get_asset(self, asset_id):
```

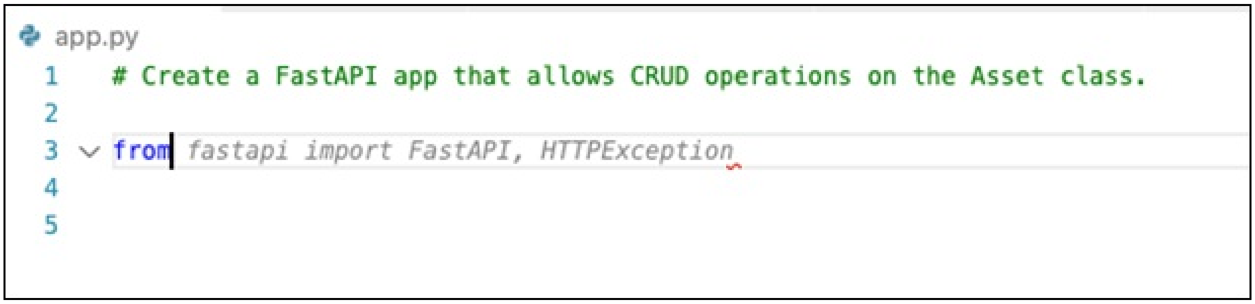
```
for asset in self.assets:
if asset.id == asset_id:
return asset
return None
def update_asset(self, asset_id, asset):
for index, asset in enumerate(self.assets):
if asset.id == asset_id:
self.assets[index] = asset
return True
return False
def delete_asset(self, asset_id):
for index, asset in enumerate(self.assets):
if asset.id == asset_id:
del self.assets[index]
return True
return False
```

Next, we will assemble the FastAPI application to run and display the Assets. We will begin with the following prompt at the beginning of the app.py class.

Listing 2.21 The starting prompt to have Copilot build our FastAPI application

```
# Create a FastAPI app that allows CRUD operations on the Asset c
```

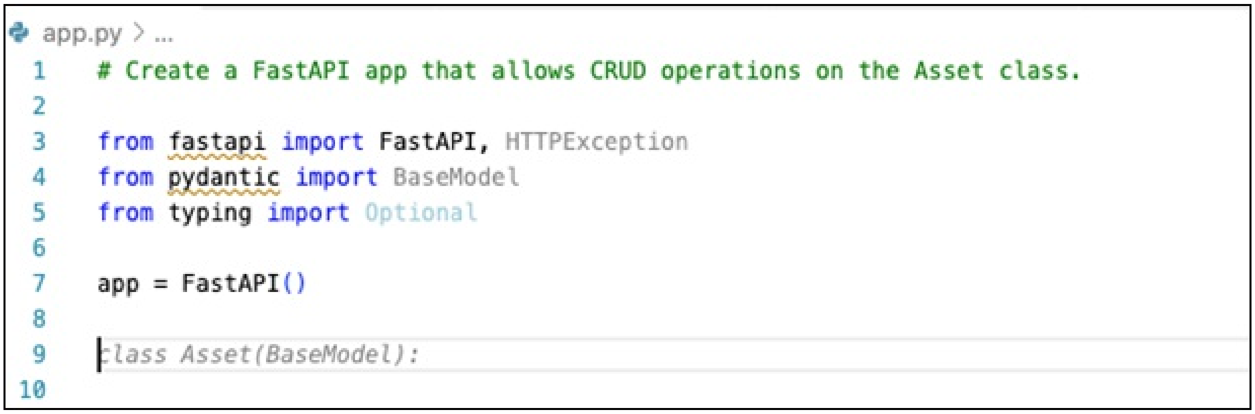
Figure 2.10 Copilot begins to automagically import the required libraries.



```
# Create a FastAPI app that allows CRUD operations on the Asset class.

from fastapi import FastAPI, HTTPException
```

Copilot will add the import statements. Something interesting happens after the definition of the FastAPI application: Copilot will attempt to define the Asset class. It is good practice to encapsulate and protect your internal model, which is the Asset class.

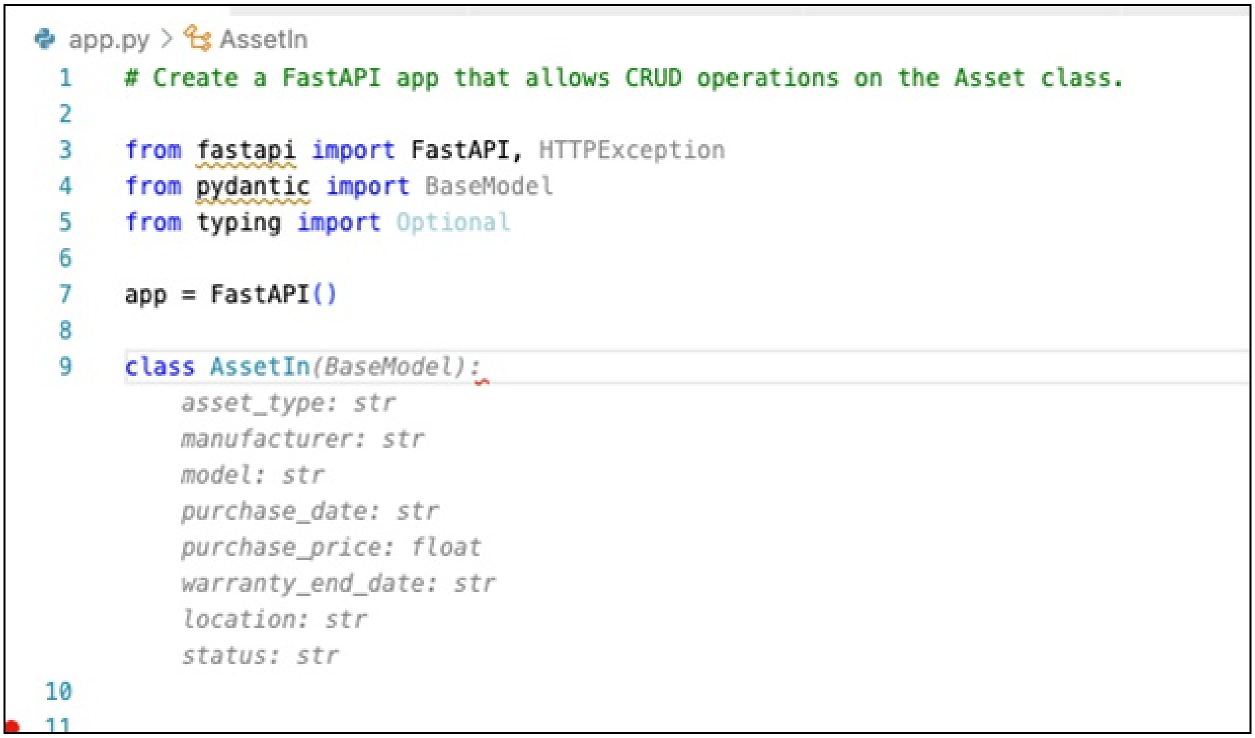
Figure 2.11 Copilot attempts to define a pydantic-based Asset class. However, we have already defined an Asset class, which represents our "internal" model of the class. We, therefore, rejected this suggestion.



```python
# Create a FastAPI app that allows CRUD operations on the Asset class.

from fastapi import FastAPI, HTTPException
from pydantic import BaseModel
from typing import Optional

app = FastAPI()

class Asset(BaseModel):

```

To protect our internal model, we will, therefore, reject the suggestion that ChatGPT provided us. Instead, we will create two classes: AssetIn and AssetOut. The Copilot-generated code closely mimics the solution that ChatGPT supplied to us. Once we type in AssetIn, Copilot will helpfully create the rest of the class for us, as it has intuited what we want to do.

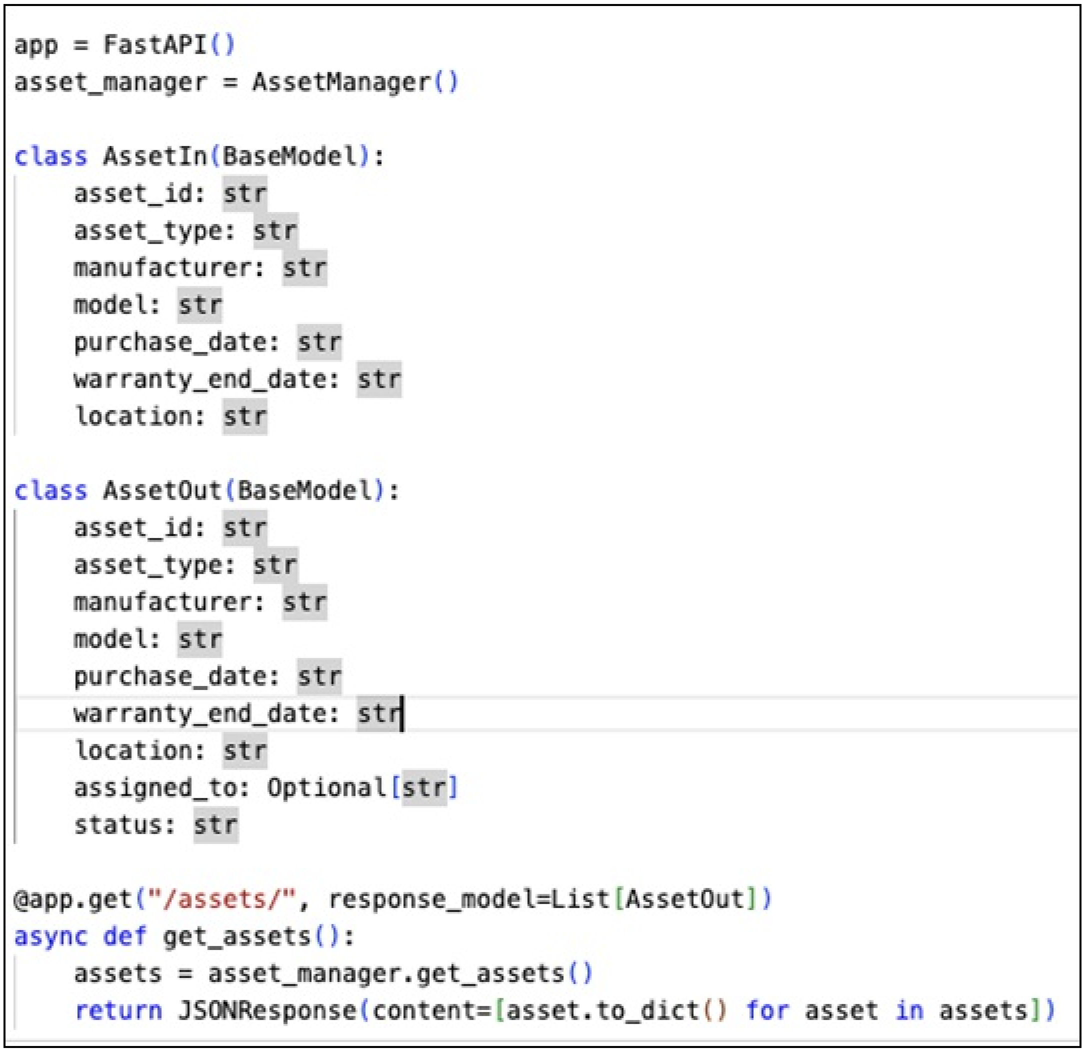
Figure 2.12 Copilot has determined that we do not want to directly expose our internal model and create a class representing the input payload from our yet-to-be-defined POST REST endpoint.



```python
# Create a FastAPI app that allows CRUD operations on the Asset class.

from fastapi import FastAPI, HTTPException
from pydantic import BaseModel
from typing import Optional

app = FastAPI()

class AssetIn(BaseModel):
    asset_type: str
    manufacturer: str
    model: str
    purchase_date: str
    purchase_price: float
    warranty_end_date: str
    location: str
    status: str


```

As we continue to define the methods in our FastAPI application, Copilot will predict the methods' implementation.

Figure 2.13 The final contents of the file app.py after Copilot has generated the implementation of all of the methods.



```
app = FastAPI()
asset_manager = AssetManager()

class AssetIn(BaseModel):
    asset_id: str
    asset_type: str
    manufacturer: str
    model: str
    purchase_date: str
    warranty_end_date: str
    location: str

class AssetOut(BaseModel):
    asset_id: str
    asset_type: str
    manufacturer: str
    model: str
    purchase_date: str
    warranty_end_date: str
    location: str
    assigned_to: Optional[str]
    status: str

@app.get("/assets/", response_model=List[AssetOut])
async def get_assets():
    assets = asset_manager.get_assets()
    return JSONResponse(content=[asset.to_dict() for asset in assets])
```

Finally, we will get Copilot to create a requirements.txt file for us. We begin with the prompt in snippet 2.21. Then, Copilot will add the version numbers as we start enumerating the libraries.

Snippet 2.21 The requirements.txt file for this project

```
# generate a requirements.txt file for this project
fastapi==0.63.0
uvicorn==0.13.4
pydantic==1.7.3
starlette==0.13.6
```

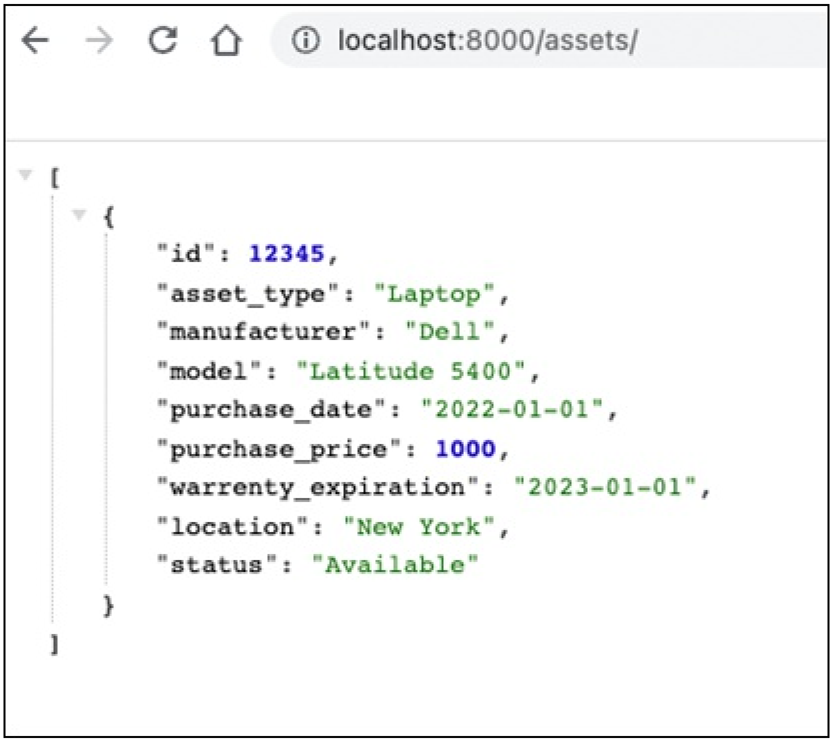
Start the application using the following command: `uvicorn app:app --reload`. Create a new Asset will the curl command in listing 2.22.

Listing 2.22 The curl command to add a new Asset to our Copilot application

```
curl -X POST -H "Content-Type: application/json" -d '{"id": "1234
```

Open your browser to http://localhost:8000/assets/ to display a new asset via the REST controller.

Figure 2.14 We created a single Asset object using the POST endpoint. This is displayed using the GET endpoint.



```
[
  {
    "id": 12345,
    "asset_type": "Laptop",
    "manufacturer": "Dell",
    "model": "Latitude 5400",
    "purchase_date": "2022-01-01",
    "purchase_price": 1000,
    "warrenty_expiration": "2023-01-01",
    "location": "New York",
    "status": "Available"
  }
]
```

To summarize, over the past two sections, we used two separate Generative AIs – ChatGPT and GitHub Copilot – to develop the same project. ChatGPT gave us considerable flexibility in how we can approach this development. We were able to switch Frameworks, though not without some trepidation. ChatGPT was able to provide suggestions and proffer advice.

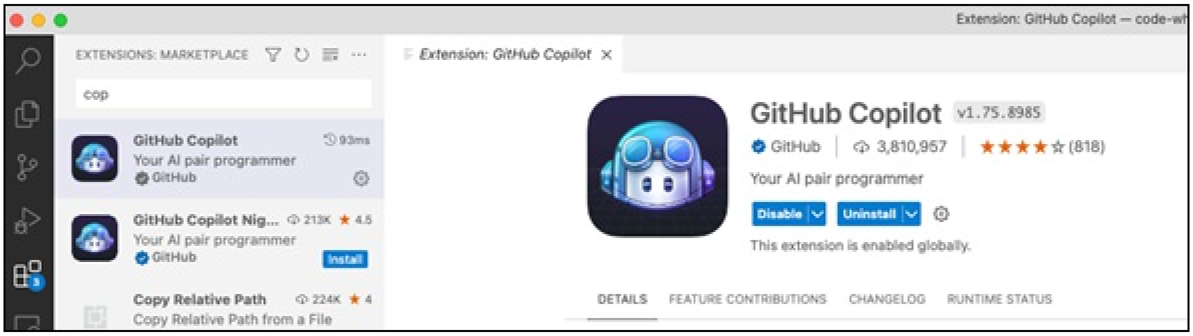
With Copilot, we used prompts to generate code for us. We needed to know upfront the implementation that we had in mind. We will now bring AWS

CodeWhisperer into this mix, and finally compare the three tools.

2.4 Let CodeWhisperer Speak Loudly

Before we begin using CodeWhisperer, we should disable Copilot. Click on the Extension table and search for Copilot. Once you have found it click the Disable button. You will need to restart the application. Once the application has restarted, you can begin to use CodeWhisperer. If you find yourself in need of assistance to install or configure CodeWhisperer, you should refer to Appendix C. Once the plugin is install and you are logged into your developer account, you should create a file called asset.py.

Figure 2.15 Before we can switch to CodeWhisperer, we must disable Copilot. We disable the extension in the Extensions tab.



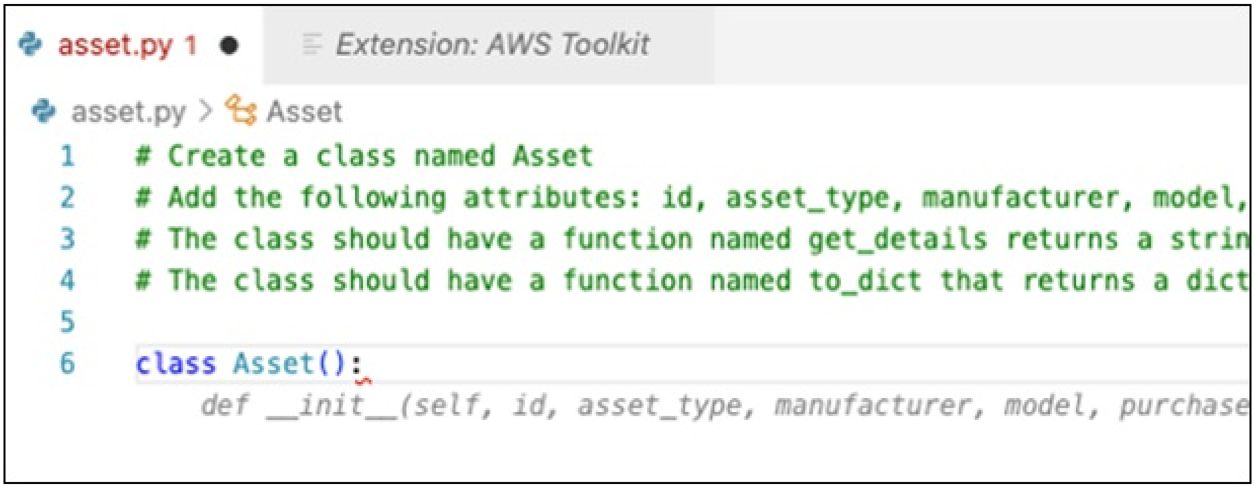
We will use the same prompt that we used for Copilot. It is reprinted here for convenience.

Listing 2.23 The prompt to have CodeWhisperer create the Asset class for us

```
# Create a class named Asset
# Add the following attributes: id, asset_type, manufacturer, mod
# The class should have a function named get_details that returns
# The class should have a function named to_dict that returns a d
```

Once we have entered this prompt and written the class definition, CodeWhisperer will begin writing the constructor.

Figure 2.16 CodeWhisperer suggests a constructor based on the prompt we entered at the beginning of the file.



```
# Create a class named Asset
# Add the following attributes: id, asset_type, manufacturer, model,
# The class should have a function named get_details returns a strin
# The class should have a function named to_dict that returns a dict

class Asset():
    def __init__(self, id, asset_type, manufacturer, model, purchase
```

We then continue to write the method definitions, and CodeWhisperer will dutifully generate the implementation.

Listing 2.24 The complete code generated by CodeWhisperer

```
# Create a class named Asset
# Add the following attributes: id, asset_type, manufacturer, mod
# The class should have a function named get_details that returns
# The class should have a function named to_dict that returns a d
class Asset():
def __init__(self, id, asset_type, manufacturer, model, purchase_
self.id = id
self.asset_type = asset_type
self.manufacturer = manufacturer
self.model = model
self.purchase_date = purchase_date
self.purchase_price = purchase_price
self.warranty_end_date = warranty_end_date
self.location = location
self.status = status
def get_details(self):
return f"{self.id}, {self.asset_type}, {self.manufacturer}, {self
def to_dict(self):
return {"id": self.id, "asset_type": self.asset_type, "manufactur
```

The final code should resemble the code created by Copilot. Extremely similar. Given this, we will skip writing the other files in the project. Completing the project will be left to a motivated reader.

Given the similarity of all of the code generated by each of these tools should naturally raise the question: how do all of these products compare? We will example that very question in the next section.

2.5 The Comparison

The first dimension we will consider is the engagement model: how we engage with AI. In the case of ChatGPT, we log into the chat website and enter prompts into a chat input box. Then, we refine our requirements in subsequent prompts. The feedback loop takes the context from the previous prompts, applies it to the current prompt, and generates output, to which the user reacts and refires. If we contrast this engagement model against the model of Copilot and CodeWhisperer, we note that the latter two tools work within an Integrated Development Environment. We do not use it outside our IDE; try as we might. The approach is not inherently inferior; it just differs.

The way that Copilot and CodeWhisperer keep you in your IDE could be seen as a benefit rather than a deficiency. In the forthcoming Copilot X release, you will get the best of both worlds: ChatGPT and GPT 4.0 all within your IDE. These tools keep you in your code without distraction for longer. Working distraction-free is one of the keys to productivity. Copilot and CodeWhisperer excel at getting out of your way; keeping you from switching contexts; keeping you free from distraction; keeping you in the flow state longer. They do this well. You engage ChatGPT in a dialog; Copilot and CodeWhisperer advise you. The dialog will take longer to work through; advice comes fast and free.

Next, we examine how the code is presented and generated. ChatGPT can create the code as a block, method, class, or project. ChatGPT reveals projects deliberatively if asked. But ChatGPT did create the project behind the scenes. ChatGPT, after all, likes to talk. With Copilot and CodeWhisperer, the code unfolds one method at a time, at least initially. As you use it more, you will notice that they can write more and more of the code for a given class. They will, however, and unfortunately, not be able to write an entire project with a tiny prompt.

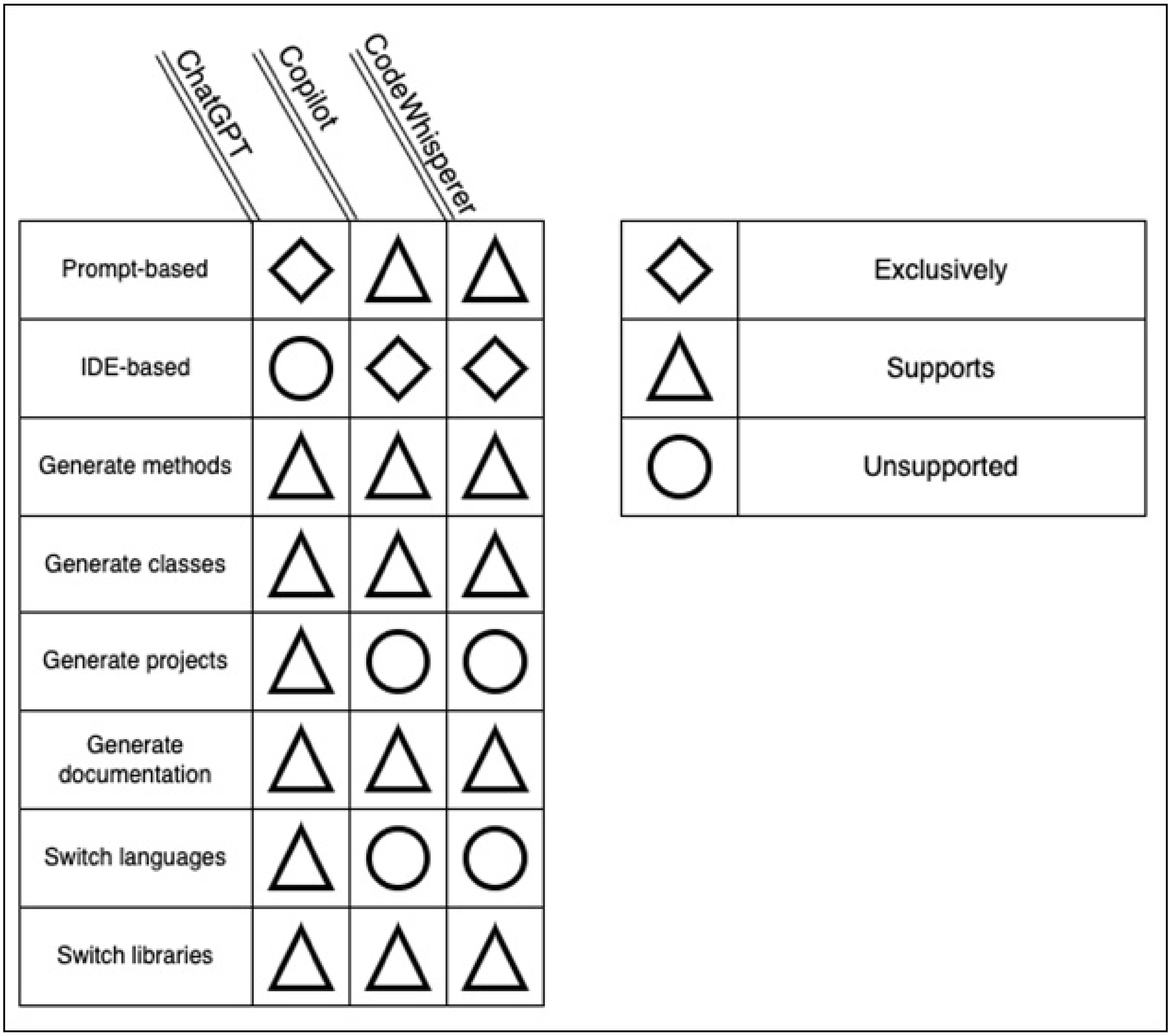
One item that they all share is their ability to respond to prompts. With ChatGPT, prompts are the only way to engage with the tool. With Copilot and CodeWhisperer, responding to prompts is not strictly necessary, but coding such prompts will make the output correspond more closely to what you initially had in mind.

Combining these factors, one might conclude that ChatGPT is an excellent choice for exploration and prototyping. However, ChatGPT can introduce unnecessary distracts, partly because you have left your IDE and are now in a Web Browser with all of the accompanying temptations that come with a browser. ChatGPT, itself, is part of the inclusion of unnecessary distractions. You will eventually fall down the proverbial rabbit hole. The tool makes it too easy, not too. Don't let that scare you off. It is a wonderful resource.

Copilot and CodeWhisperer require that you have a desired outcome in mind. Therefore, these tools are perfect for when you want to head down, coding with precise requirements and tight deadlines. Copilot and CodeWhisperer work best when you know the language and the framework. They can automate much of the drudgery, allowing you to focus on the business requirements, which add value and are likely why you are writing the software in the first place.

The following chart briefly summarizes these benefits and limitations of all three AIs:

Figure 2.17 A comparison of the positives and negatives of ChatGPT, Copilot, and CodeWhisperer.



	ChatGPT	Copilot	CodeWhisperer
Prompt-based	◇	△	△
IDE-based	○	◇	◇
Generate methods	△	△	△
Generate classes	△	△	△
Generate projects	△	○	○
Generate documentation	△	△	△
Switch languages	△	○	○
Switch libraries	△	△	△

◇	Exclusively
△	Supports
○	Unsupported

We went through a lot in this chapter, revisiting the same code several times. However, writing three versions of the same subsection of the ITAM system was foundational, allowing us to recognize when it would make sense to use ChatGPT as opposed to when to use the other IDE-focused tools such as Copilot and CodeWhisperer. In subsequent chapters, we will leverage this knowledge, choosing the most suitable tool.

One final note, these tools work best when they work together. ChatGPT is an excellent tool for example and structure. Copilot and CodeWhisperer provide you with the ability to extend and customize the code. This makes the forthcoming Copilot X release so exciting: it is a marriage of both of these approaches.

2.6 Summary

- An Information Technology Asset Management (ITAM) system is a tool to manage and track hardware devices, software licenses, and other IT-related components throughout their lifecycle. These systems allow companies to control costs, manage vendor relationships, and ensure licensing and regulatory compliance.
- In this chapter, we explored using ChatGPT, GitHub Copilot, and AWS CodeWhisperer by writing the same code in all three tools. Writing the same code thrice allowed us to compare the features and contrast the differences in usage: how and when to use each.
- ChatGPT is a prompt-based Generative AI that engages the user in a dialog that helps them explore ideas to aid in the design and development of entire projects. In addition, ChatGPT artfully generates documentation for each method that it writes. One of the reasons that we began the chapter using it is because it helped define a template we used throughout the remainder of the chapter. It is a fascinating product, one that can lead to unnecessary albeit enjoyable distractions.
- Copilot and CodeWhisperer are heads-down tools, working best when you know what you want to do and need some advice on how best to get it done. How one engages with these tools is remarkably similar, as are the results.
- ChatGPT (as of this writing) does not support development within an IDE. However, unlike GitHub Copilot and AWS CodeWhisperer, it can produce entire projects and easily translate code from one programming language to another. GitHub Copilot and AWS CodeWhisperer take hints from your comments to infer what code you want to be written. With ChatGPT, you explicitly write prompts that ChatGPT will use to create the code.

3 Designing, Then Building

This chapter covers

- Prototyping potential designs with ChatGPT
- Documenting our architecture in Mermaid
- Completing our design with ChatGPT

Now that we have an intuition around when to use Generative AI, we will start to design, explore, and document our application's architecture. Laying out some of the critical components upfront is beneficial in several ways. For example, it allows us to delegate some of the design to sub-architects or some of the development to other team members. Designing up front will also help us clarify our thinking about the implementation, allowing us to anticipate and avoid some pitfalls. Finally, capturing the design as documentation enables us to justify our crucial design decisions, communicating our intent to our future selves, our stakeholders, and those who may inherit the project.

First, let’s solicit ChatGPT’s opinion on how best to design this project and see what solution it can come up with.

3.1 Petitioning for Aid

We will begin with a prompt, in a new session, that will outline our requirements. We should be able to construct an elaborate prompt given that we spent much of the last chapter thinking about our requirements and should have an excellent idea about what is needed.

Listing 3.1 A prompt featuring the requirements for our system

```
Please take the following criteria for an information technology

Asset Inventory: The system should be able to maintain a complete
Asset Tracking: The system should be able to track asset location

Asset Lifecycle Management: The system should be able to manage t

Asset Allocation: The system should be able to allocate assets to

Asset Reporting: The system should be able to generate reports on

Integration: The system should be able to integrate with other IT
based access control, and data encryption.
Scalability: The system should be scalable to accommodate changes
  User-Friendly Interface: The system should have a user-
friendly interface that is easy to navigate, search, and use. It

Accessibility: The system should be accessible from different dev
```

After some deliberation, ChatGPT will return the high-level steps of what is needed to be done to develop this project. Oh, who are we kidding, it returns these steps immediately. These suggestions are helpful, but we are really here to get past the empty file problem: we want to get started designing. We want prose and diagrams to help us understand the system at the component level. We need tasks not a recipe. We should ask ChatGPT to show its work.

Listing 3.2 A prompt to have ChatGPT to show its work

```
Please show the package structure
```

Unbeknownst to us, ChatGPT created a project. Having asked it what it created; it will return the following diagram (slightly abridged).

Listing 3.3 The package structure for the Asset Management System that ChatGPT created

```
asset_management_system/ ├── app/ │   ├── __init__.py │   ├── api/
│   │   ├── __init__.py │   │   ├── asset.py │   │   ├── user.py
│   │   └── ... │   ├── auth/ │   │   ├── __init__.py
│   │   ├── login.py │   │   ├── register.py │   │   └── ...
│   ├── models/ │   │   ├── __init__.py │   │   ├── asset.py
│   │   ├── user.py │   │   └── ... │   ├── views/
│   │   ├── __init__.py │   │   ├── asset.py │   │   ├── user.py
│   │   └── ... ├── requirements.txt └── run.py
```

You should note that ChatGPT (and therefore you) has access to its code: all we need do ask ChatGPT for the code!

Listing 3.4 A prompt to have ChatGPT to show its (detailed) work

```
please show the source code for asset_management_system/models/as
```

ChatGPT will return the source code for this class. Note that it is using Flask and SQLAlchemy, so the domain model is a little muddled. Next time, we should ask it to use our preferred application server and keep the domain model clean. These are additional data points for us as well. Now we have an idea of what we don’t want.

Listing 3.5 The example source code for the Asset model in asset_management_system/models/asset.py:

```
from datetime import datetime
from asset_management_system import db  class Asset(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(80), nullable=False)
    description = db.Column(db.String(255))
    category = db.Column(db.String(80))
    purchase_date = db.Column(db.DateTime, default=datetime.utcno
    purchase_cost = db.Column(db.Float)
    warranty_expiry = db.Column(db.Date)
    lease_expiry = db.Column(db.Date)
    status = db.Column(db.String(80), default="Available")
    location = db.Column(db.String(80), default="IT Department")
    assigned_to = db.Column(db.String(80))
    assigned_date = db.Column(db.DateTime)
    retired_date = db.Column(db.DateTime)
    created_date = db.Column(db.DateTime, default=datetime.utcnow
    updated_date = db.Column(db.DateTime, default=datetime.utcnow
      def __repr__(self):        return '<Asset %r>' % self.name
```

Now that we have a sense for how ChatGPT would tackle the problem, let’s get a second opinion. To do this we will explore existing projects and libraries. We will also use ChatGPT to do this investigation.

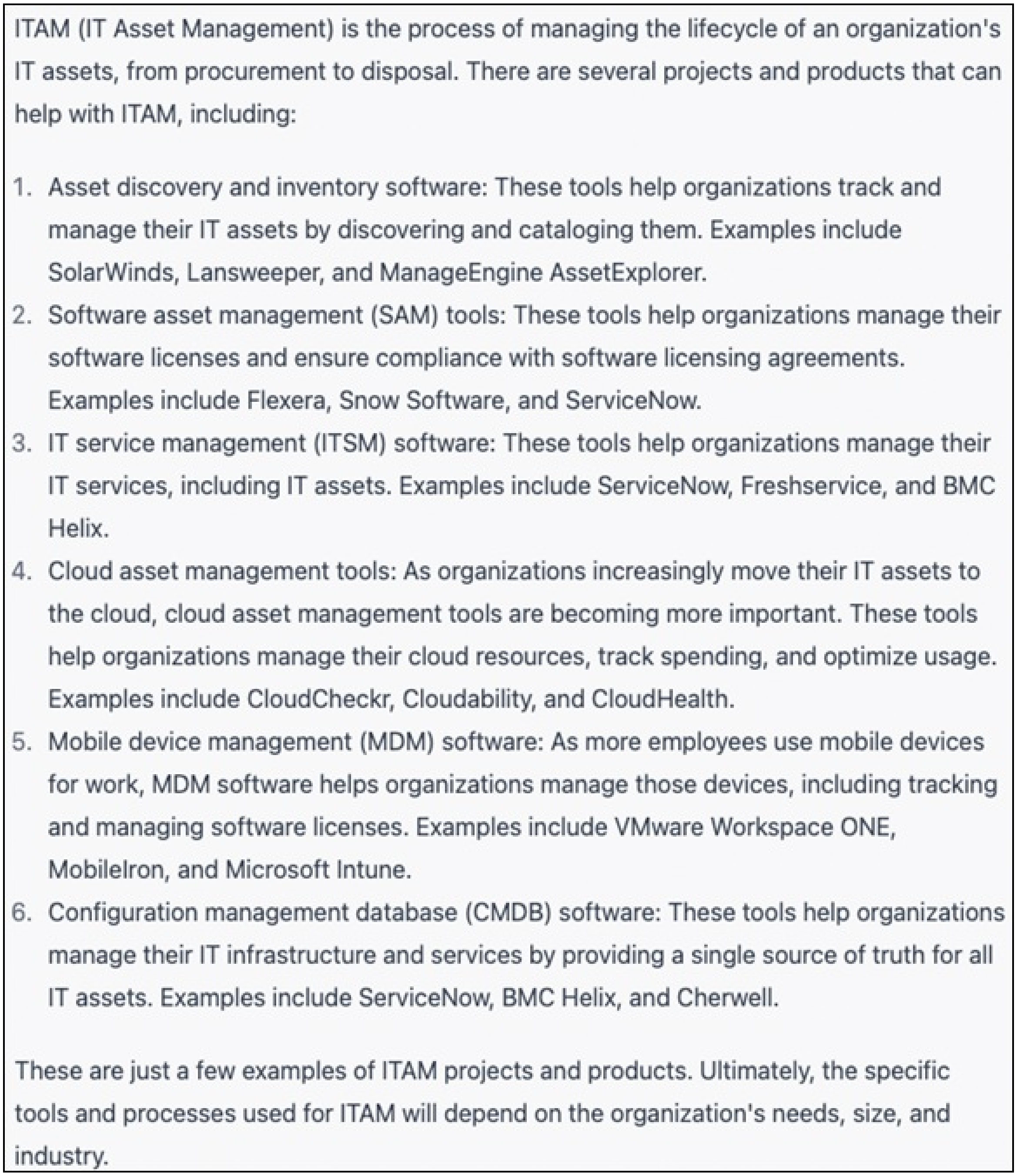
3.2 Surveying the Landscape

If we have never used or seen an Information Technology Asset Management (ITAM) system before, we would want to understand what it is, what it should do, and how to implement such a system. As we experienced in the last chapter, this is a task to which ChatGPT is well suited. GitHub Copilot and CodeWhisperer are excellent tools when the problem space and known, when you are ready for head-down highly productive coding. ChatGPT, on the other hand, can be used to explore this problem space, to prototype interactively, and to design and document your solution. A good first step in exploring the problem space is to examine current open-source projects. We can use these "out of the box" projects or leverage their design to inspire ours. Using ChatGPT, we can quickly navigate the open-source world and zero in on any projects that might be analogous to what we are attempting to build. We can start with the prompt in listing 3.1.

Listing 3.6 A prompt to find out what ITAM projects are out there

```
What are some examples of ITAM projects or products?
```

ChatGPT responds with several examples, summarizing their core features. However, since we are looking for inspiration for our development project, we should start to refine this list to only those for which the code is available; that is, which are open-source projects?



ITAM (IT Asset Management) is the process of managing the lifecycle of an organization's IT assets, from procurement to disposal. There are several projects and products that can help with ITAM, including:

1. Asset discovery and inventory software: These tools help organizations track and manage their IT assets by discovering and cataloging them. Examples include SolarWinds, Lansweeper, and ManageEngine AssetExplorer.
2. Software asset management (SAM) tools: These tools help organizations manage their software licenses and ensure compliance with software licensing agreements. Examples include Flexera, Snow Software, and ServiceNow.
3. IT service management (ITSM) software: These tools help organizations manage their IT services, including IT assets. Examples include ServiceNow, Freshservice, and BMC Helix.
4. Cloud asset management tools: As organizations increasingly move their IT assets to the cloud, cloud asset management tools are becoming more important. These tools help organizations manage their cloud resources, track spending, and optimize usage. Examples include CloudCheckr, Cloudability, and CloudHealth.
5. Mobile device management (MDM) software: As more employees use mobile devices for work, MDM software helps organizations manage those devices, including tracking and managing software licenses. Examples include VMware Workspace ONE, MobileIron, and Microsoft Intune.
6. Configuration management database (CMDB) software: These tools help organizations manage their IT infrastructure and services by providing a single source of truth for all IT assets. Examples include ServiceNow, BMC Helix, and Cherwell.

These are just a few examples of ITAM projects and products. Ultimately, the specific tools and processes used for ITAM will depend on the organization's needs, size, and industry.

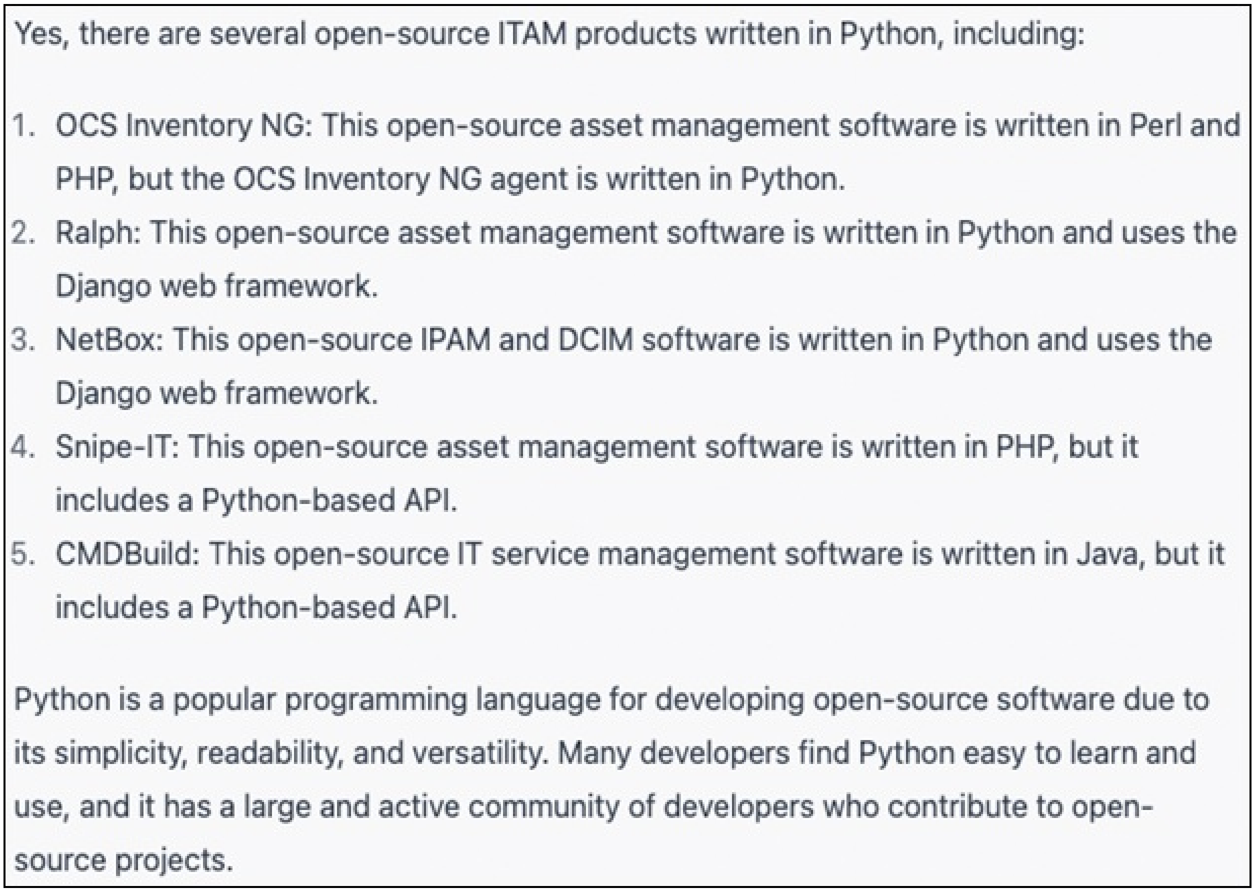
Figure 3.1 ChatGPT presents a list of attributes of the features of an ITAM and several products and projects that possess those features.

Next, we will start the refinement process. Given that the point of doing this analysis and exploration of current ITAM systems is that we can see what features are needed in the software and how we might implement these features, we will want only to return projects for which the source code is available. Therefore, let's get a list of only open-source projects.

Listing 3.7 A prompt to find out which ITAM projects are open-source

```
Are any of these products open-source?
```

One of the most exciting and helpful features of ChatGPT is that it carries context forward; it understands that in this context, "any" means any of the projects, as mentioned earlier. ChatGPT responds with a listing of open-source projects.



Yes, there are several open-source ITAM products written in Python, including:

1. OCS Inventory NG: This open-source asset management software is written in Perl and PHP, but the OCS Inventory NG agent is written in Python.
2. Ralph: This open-source asset management software is written in Python and uses the Django web framework.
3. NetBox: This open-source IPAM and DCIM software is written in Python and uses the Django web framework.
4. Snipe-IT: This open-source asset management software is written in PHP, but it includes a Python-based API.
5. CMDBuild: This open-source IT service management software is written in Java, but it includes a Python-based API.

Python is a popular programming language for developing open-source software due to its simplicity, readability, and versatility. Many developers find Python easy to learn and use, and it has a large and active community of developers who contribute to open-source projects.

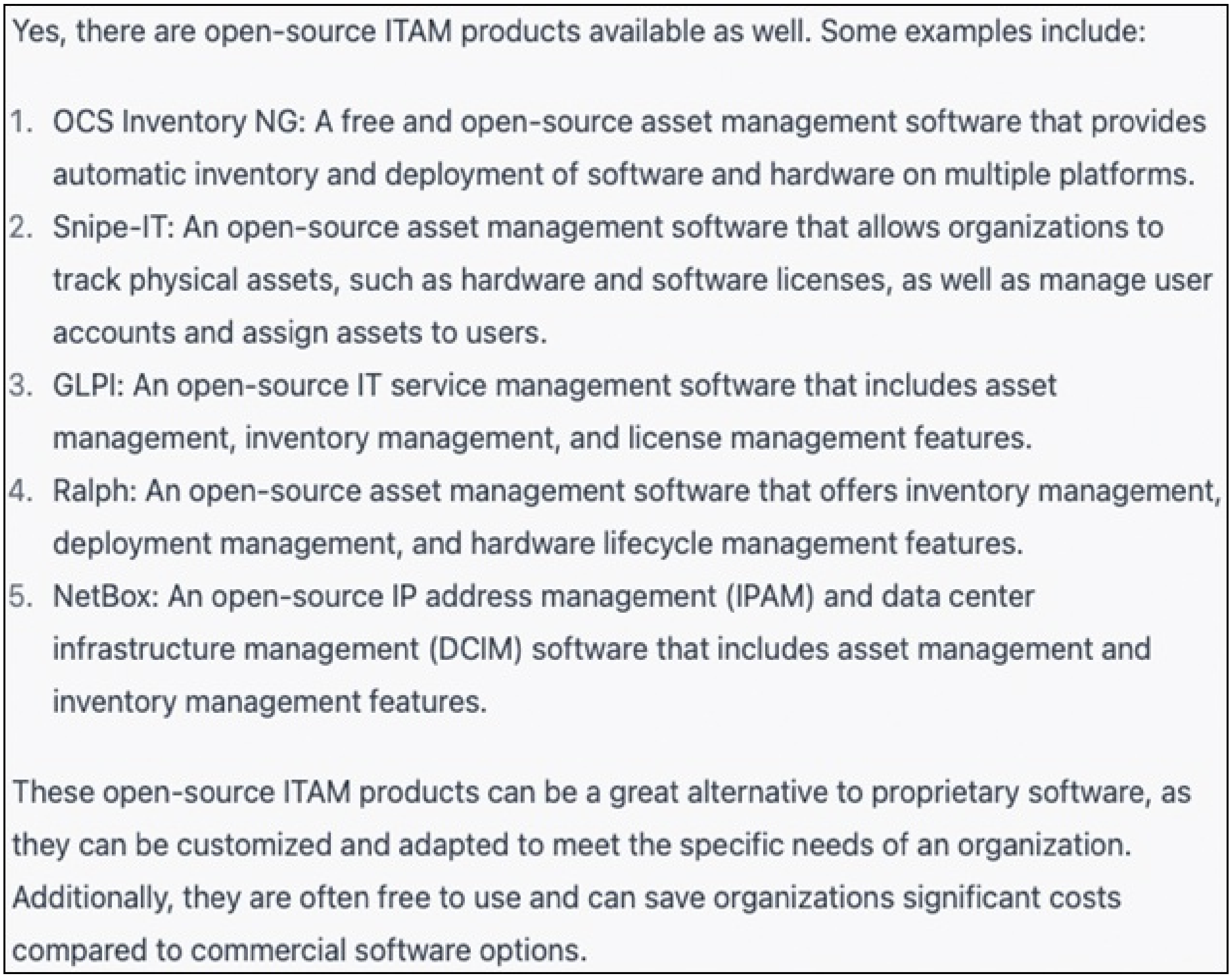
Figure 3.2 ChatGPT returns a list of open-source ITAM projects.

We will continue the refinement process. We intend to implement our system in Python, so we are only interested in projects that feature Python as a means of interacting with the system.

Listing 3.8 A prompt to find out which ITAM projects are written in Python

```
Are any of these written in Python?
```

Based on our prompt, ChatGPT will widdle down this list to only those projects written in Python. It returns five projects. We will assess each project and determine which one we should explore further.



Yes, there are open-source ITAM products available as well. Some examples include:

1. OCS Inventory NG: A free and open-source asset management software that provides automatic inventory and deployment of software and hardware on multiple platforms.
2. Snipe-IT: An open-source asset management software that allows organizations to track physical assets, such as hardware and software licenses, as well as manage user accounts and assign assets to users.
3. GLPI: An open-source IT service management software that includes asset management, inventory management, and license management features.
4. Ralph: An open-source asset management software that offers inventory management, deployment management, and hardware lifecycle management features.
5. NetBox: An open-source IP address management (IPAM) and data center infrastructure management (DCIM) software that includes asset management and inventory management features.

These open-source ITAM products can be a great alternative to proprietary software, as they can be customized and adapted to meet the specific needs of an organization. Additionally, they are often free to use and can save organizations significant costs compared to commercial software options.

Figure 3.3 ChatGPT informs us of four open-source projects with Python components.

Of this list, Ralph appears to be the most promising. Snipe-IT and Open-AudIT are written in PHP with a Python-based API that we could interact with; however, we want a project written in Python. NetBox is not an ITAM but an IP Address Management (IPAM) system. Finally, CMDBuild is written in Java. We want to build an IT Asset Management system in Python. Therefore, Ralph would appear to be the closest match to our use case.

In the real world

If we had to implement an ITAM system in the real world, we would assess how well each of these products (commercial and open source) fit our different use cases. If, for example, Ralph could satisfy most or all of those use cases, we would just stand up a new instance; or in the case of a commercial product, we would acquire a license. We would perform this analysis because we get paid to deliver business value, not write code. If we can take something off the shelf and just use it, then we have delivered value quickly, likely quicker than we would have if we had developed it ourselves.

Now that we have discovered a similar project to the one, we are looking to build (a subset of), we can start exploring it, reviewing its design, and examining its source code. We will draw inspiration from this source code and design, borrow liberally, and change where appropriate. Let's begin with an overview of the system.

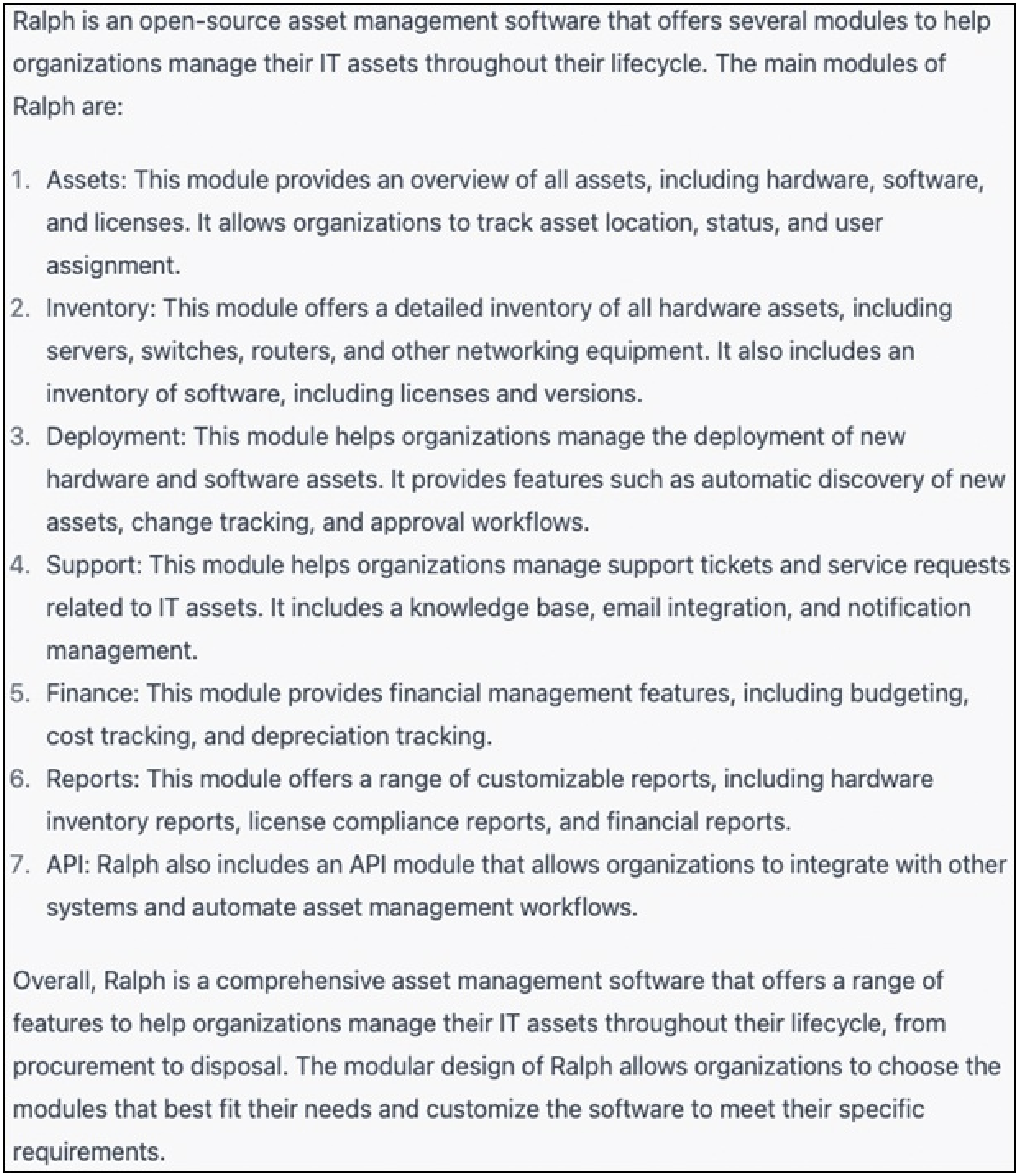
3.3 Exploring a Library

First, we will ask ChatGPT to describe Ralph's overall structure. This structure might inspire us as we develop our architecture. Alternatively, it might caution us on how not to structure our code.

Listing 3.9 A prompt to find out the structure of the open source project Ralph

```
Please describe the modules of Ralph.
```

ChatGPT will return a listing of high-level modules of Ralph. One thing to note, at the time of this writing, ChatGPT will produce the design of Ralph as of 2021. ChatGPT's training data cut-off data (according to ChatGPT itself) is September 2021. Therefore, any subsequent changes to the design post-September 2021 would not be available in ChatGPT. This will all change in the future, when GPT has internet access and extensions. For now, however, we must accept this limitation.



Ralph is an open-source asset management software that offers several modules to help organizations manage their IT assets throughout their lifecycle. The main modules of Ralph are:

1. Assets: This module provides an overview of all assets, including hardware, software, and licenses. It allows organizations to track asset location, status, and user assignment.
2. Inventory: This module offers a detailed inventory of all hardware assets, including servers, switches, routers, and other networking equipment. It also includes an inventory of software, including licenses and versions.
3. Deployment: This module helps organizations manage the deployment of new hardware and software assets. It provides features such as automatic discovery of new assets, change tracking, and approval workflows.
4. Support: This module helps organizations manage support tickets and service requests related to IT assets. It includes a knowledge base, email integration, and notification management.
5. Finance: This module provides financial management features, including budgeting, cost tracking, and depreciation tracking.
6. Reports: This module offers a range of customizable reports, including hardware inventory reports, license compliance reports, and financial reports.
7. API: Ralph also includes an API module that allows organizations to integrate with other systems and automate asset management workflows.

Overall, Ralph is a comprehensive asset management software that offers a range of features to help organizations manage their IT assets throughout their lifecycle, from procurement to disposal. The modular design of Ralph allows organizations to choose the modules that best fit their needs and customize the software to meet their specific requirements.

Figure 3.4 ChatGPT explains the high-level module layout of the Ralph open-source project

The Asset and Inventory modules seem like an excellent place to start. Next, we will ask ChatGPT to present a Mermaid diagram of each module. Let us first begin with Asset.

Listing 3.10 A prompt to find out the classes in the Asset module of Ralph

```
Please show me a class diagram in Mermaid for all classes in the
```

Mermaid

Mermaid is a modern, open-source, stripped-down diagram definition language invented by Knut Sveidqvist. It was released in 2015. It allows users to create various diagrams, including flowcharts, sequences, class, Gantt charts, etc. Since it is text-based it is easy to copy and paste into designers. Its syntax is very easy to read and understand with little formal to no formal training (unlike say UML).

ChatGPT will return a Mermaid document outlining the classes, their attributes, and their relationships. First, let's walk through a class definition in this Mermaid document. The first line of listing 3.11 instructs Mermaid that we are defining a class diagram type rather than the other types it supports. The second line represents the class (in this case, Asset). The following lines enumerate the fields of the class. The minus means that it is a private attribute. The second to last line is the definition of a public method. We know that it is public; the plus denotes a public method.

Listing 3.11 The Asset class definition

```
classDiagram     class Asset {        - name: string
        - status: string        - model: Model
        - category: Category        - manufacturer: Manufacturer
        - sn: string        - barcode: string
        - price: decimal        - purchase_date: date
        - order_no: string        - delivery_date: date
        - invoice_date: date        - invoice_no: string
        - provider: Provider        - budget_info: BudgetInfo
        - warehouse: Warehouse        - owner: User
        - location: Location        - asset_info: AssetInfo[]
        - attachments: Attachment[]        + get_name(): string
    }
```

Next, we will look at how to define relationships in Mermaid. The double dash with the arrow denotes a unidirectional association. In the following example, an Asset has a Model. Additionally, the Asset has a Category.

Listing 3.12 The relationship definitions in a Mermaid document

```
    Asset --> Model     Asset --> Category
```

With the complete Mermaid diagram, we can use a modeling tool to import and visualize the class diagram. Visualizing it this way will make it easier to understand. After all, a (diagram) is worth a thousand words, as they say. You should be able to plug the text into https://mermaid.live to generate and visualize the diagram.

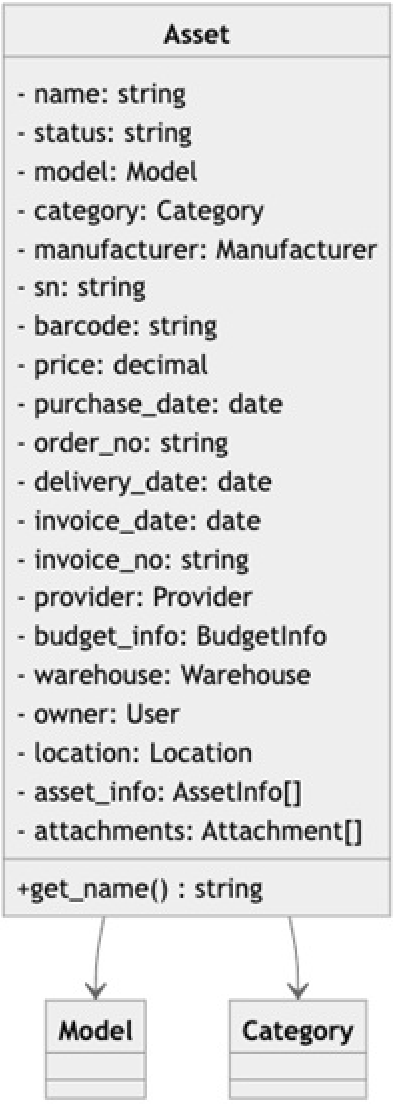


Figure 3.5 The visualization of the Asset class using a Mermaid class diagram.

Mermaid strikes the right balance between expressiveness and brevity, making it an ideal modeling language whether you are in a hurry or not. But your mileage may vary.

Next, let's turn our attention to the Inventory module. We can ask ChatGPT to produce a document similar to the Asset diagram we had previously requested. We will jump ahead to the visualization of this document.

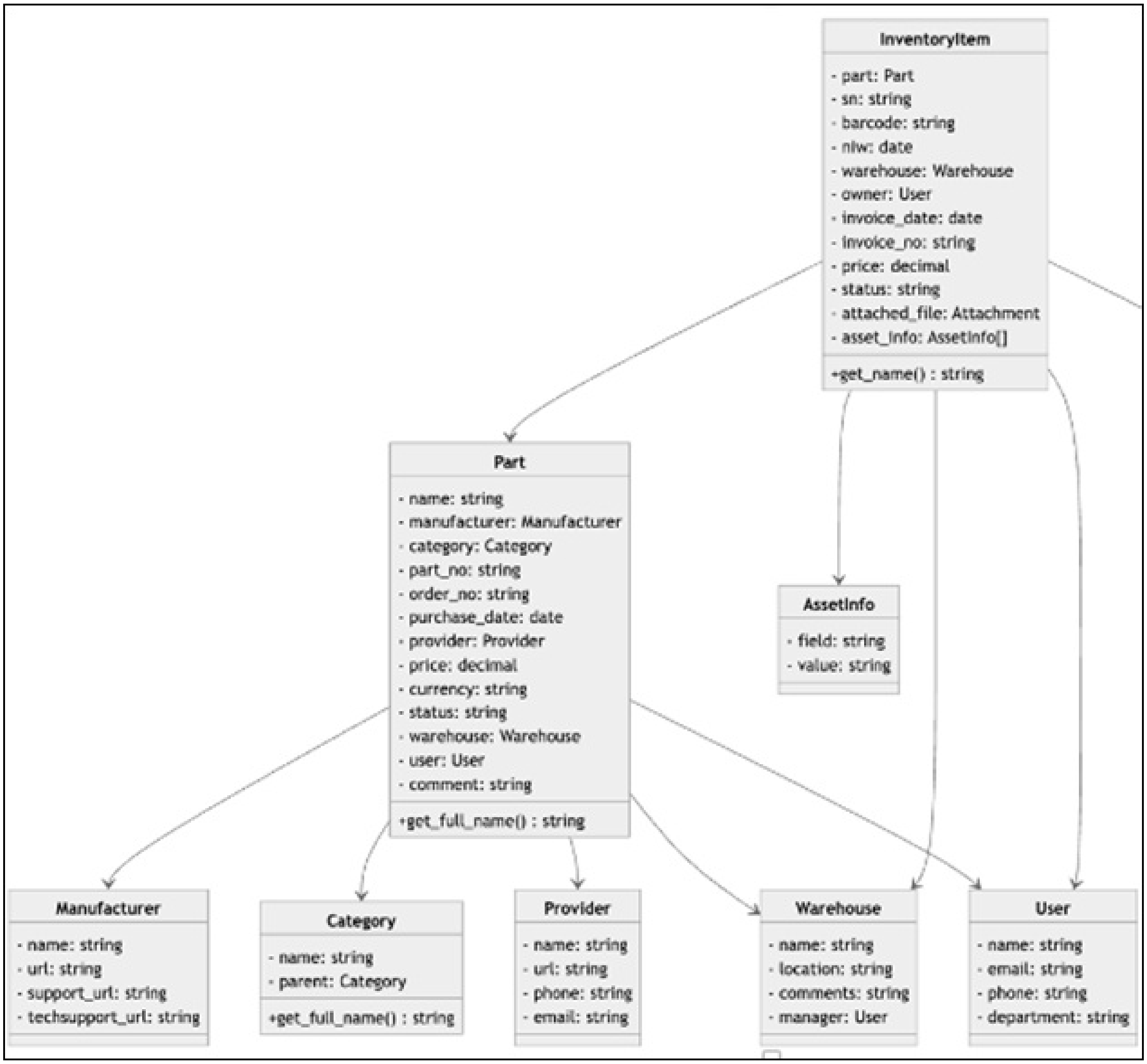


Figure 3.6 The visualization of the Inventory package using a Mermaid class diagram.

The visualization of the Inventory model clarifies that the Inventory module, while important to the Ralph project, is extraneous to our effort of building a hardware-focused ITAM. We are interested in tracking the totality of our assets, whole units; we are not necessarily interested in tracking every video card or memory module: just the entire server. We, therefore, will set this module aside.

Next, we will drill into the Asset class since this appears to be the root of the

Asset module. Let's ask ChatGPT to show us the class.

Listing 3.13 Prompt to have ChatGPT show the use of the source code for the Asset class

```
  Please show me the source code for the Asset class.
```

ChatGPT will return the source code for the Asset class. For the sake of brevity, we will not show the import statements. Additionally, we will just examine a few lines in this code block. It is immediately apparent that this class has a lot of Django-specific code. The model attribute, for instance, looks up the model object from the database. The same is true of the category.

Make ChatGPT continue Occasionally, ChatGPT will stop its output mid-sentence or mid-stream. This is due an output limitation built into the design of ChatGPT. It appears that you cannot get around this limitation by telling ChatGPT to ignore this limitation (something that you can do for certain system constraints). However, you can tell it to "continue" or "go on." It will resume the output where it left off. If it discontinues within a code block, you should specify exactly where it should resume the output.

Listing 3.14 The abridged source code to the Asset class

```
class Asset(TransitionModelMixin, NamedMixin, AdminAbsoluteUrlMix
    """    A hardware or software asset    """
    name = models.CharField(max_length=255, unique=True)
    status = models.CharField(choices=STATUS_CHOICES, default=STA
    model = models.ForeignKey(        'assets.AssetModel',
        verbose_name=_('model'),        related_name='assets',
        on_delete=models.PROTECT,        null=True,
        blank=True,    )    category = models.ForeignKey(
        'assets.Category',        verbose_name=_('category'),
        related_name='assets',        on_delete=models.PROTECT,
    )
```

We do not want to use Django in our project. We want our application to be API first, requiring little of the additional bells and whistles that Django can provide. Also, we would prefer a clean domain model. Django is opinionated, recommending that you modify your domain model classes to inherit from a Django base class. The base class defines ways that your domain model classes will interact with the database for persistence. We would prefer handling the persistence ourselves.Therefore, let's ask ChatGPT to remove Django from our project entirely.

Listing 3.15 Prompt to have ChatGPT remove Django and return the Asset class code.

```
  Take the open-
source Ralph project, remove Django, and show me the source code
```

The output produced by ChatGPT will be much cleaner. Here is the abridged result for comparison against the Django definition. The following listing displays only the constructor, as other class details are irrelevant now.

Listing 3.16 The Asset class source code without the Django-specific functionality

```
class Asset:
    def __init__(self, name, status, model, category, manufacture
        self.name = name        self.status = status
        self.model = model        self.category = category
        self.manufacturer = manufacturer        self.sn = sn
        self.barcode = barcode        self.price = price
        self.purchase_date = purchase_date
        self.order_no = order_no
        self.delivery_date = delivery_date
        self.invoice_date = invoice_date
        self.invoice_no = invoice_no
        self.provider = provider
        self.budget_info = budget_info
        self.warehouse = warehouse        self.owner = owner
        self.location = location
        self.asset_info = asset_info or []
        self.attachments = attachments or []
```

We would notice no changes if we asked ChatGPT to recreate the Mermaid class diagram. We would not see any changes because the Django-specific features were encapsulated within the classes.

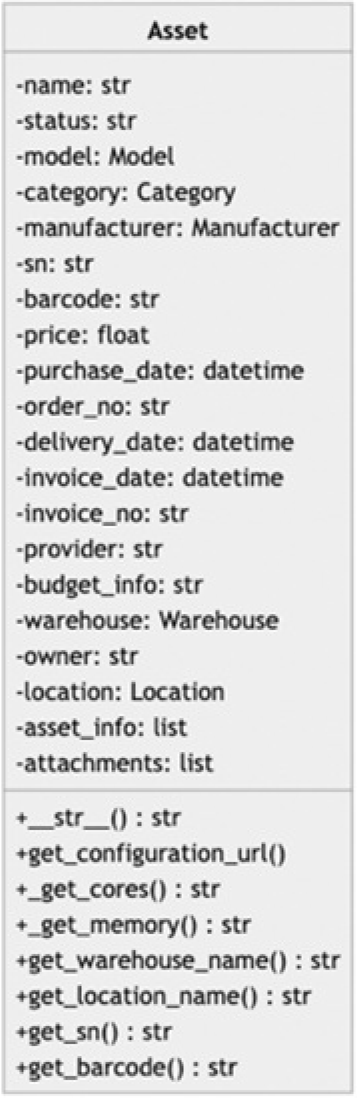


Figure 3.7 An updated Mermaid class diagram for the Asset class. The class appears unchanged from the previous version

3.4 Documenting Your Architecture

In the last section, we have explored the Ralph open-source project and understand how the project fits together, and we can begin our design. We will work with ChatGPT iteratively to help us with our design and documentation. Let's start with a brand-new chat window. The new chat session will ensure that the context is clear; that none of our previous prompts will influence our new design.

First, we will ask ChatGPT to design the initial application design. We will use the following prompt to do so.

Listing 3.17 A prompt for ChatGPT to design our initial application stub

```
I would like to build an ITAM project, written in Python. It will
```

Hexagonal Architecture

Hexagonal Architecture, also known as the Ports and Adapters pattern, is an architectural pattern that aims to create a clear separation between an application's core logic and its interaction with external systems, such as databases, user interfaces, and third-party services. This separation of concerns helps to achieve a more maintainable, flexible, and testable application.

The main components of Hexagonal Architecture are:

Domain Model: This represents the core business logic of the application, including entities, value objects, and domain services. The Domain Model is isolated from any external systems or technologies and should not depend on any specific implementation details, like databases or APIs.

Ports: Ports are interfaces that define the contract between the Domain Model and the external systems. They represent the input and output boundaries of the application. There are two types of ports: Primary and Secondary.

Primary (Driven) Ports: These define the use cases that the application exposes to the external systems. They represent the API that external systems can use to interact with the application.

Secondary (Driving) Ports: These define the contracts that the application expects from external systems, like data repositories or external services.

Adapters: Adapters are the implementations of the Ports. They handle the communication between the Domain Model and the external systems, converting the data and protocols used by external systems into a format that the Domain Model can understand. There are two types of adapters: Primary and Secondary.

Primary (Driven) Adapters: These implement the Primary Ports and are responsible for receiving input from external systems and passing it to the Domain Model.

Secondary (Driving) Adapters: These implement the Secondary Ports and are responsible for interacting with external systems on behalf of the Domain Model.

In Hexagonal Architecture, the Domain Model is at the center (the "hexagon"), surrounded by the Ports and Adapters. The key idea is that any interaction between the Domain Model and external systems must pass through the Ports and Adapters. This architecture allows for easy swapping of external systems, simplified testing by using mock implementations of Ports, and clear separation of concerns, promoting a more maintainable application.

In response, ChatGPT creates this peculiar class structure. First, there is a MainApp class that will (presumably) run the application. This class uses the FastAPIAdapter, which accepts and handles REST calls. The FastAPIAdapter would invoke the methods of the HardwareService.

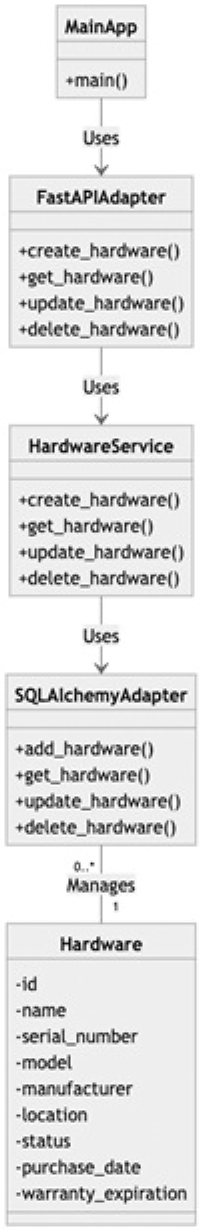


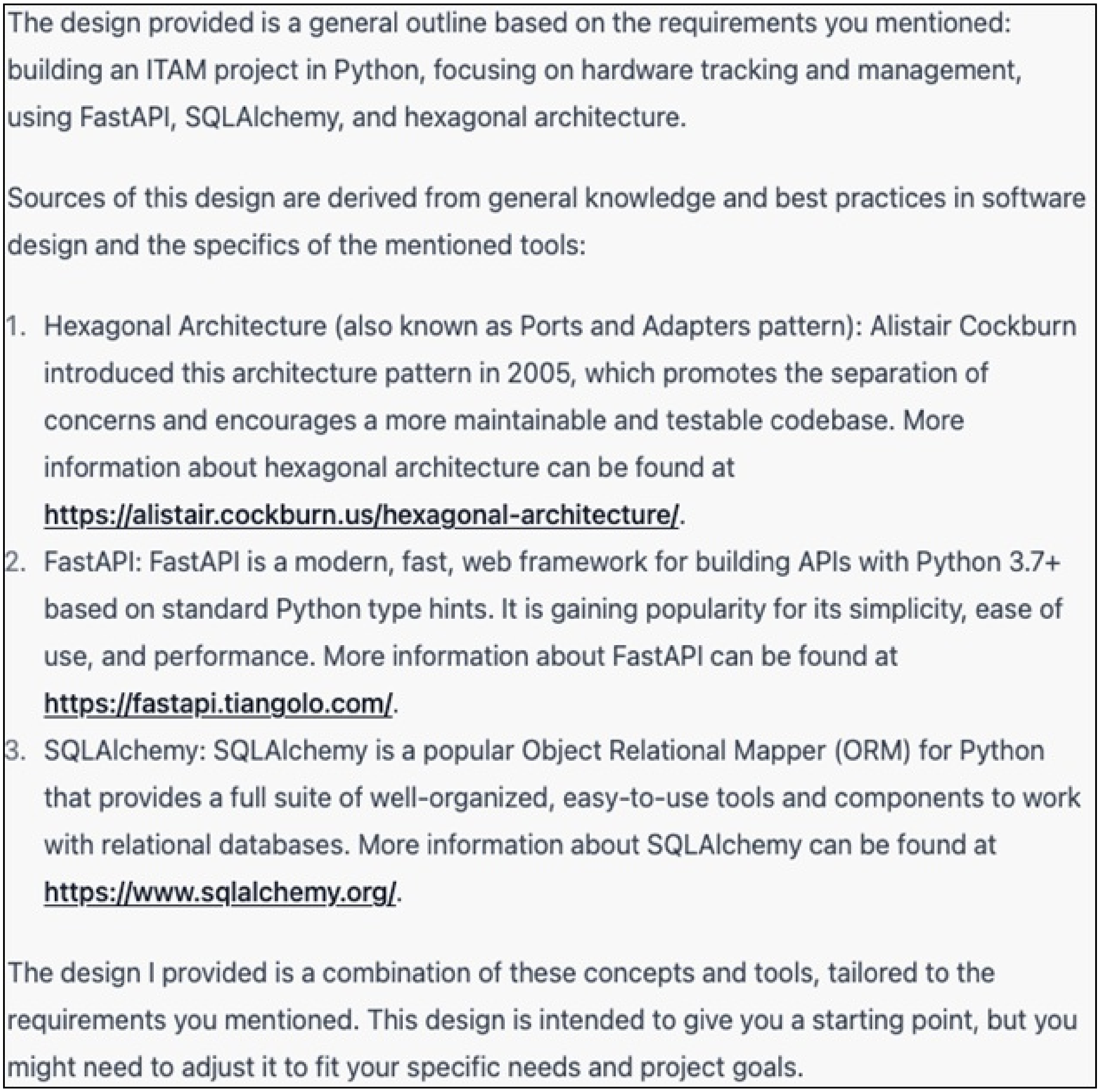
Figure 3.8 The straightforward design for the ITAM system suggested to us by ChatGPT

We can attempt to ask ChatGPT why it returned this design to us. Getting ChatGPT to cite actual sources is a feat to prompt engineering excellence as ChatGPT tries to speak only general terms. However, let’s try to see if we can get it to be specific using the following prompt.

Listing 3.18 A prompt to ask ChatGPT for inspiration for its design

```
Please provide the sources of this specific design.
```

ChatGPT's response to this prompt is very general. It cites best practices and the origins of hexagonal architecture. However, it does not tell us specifics about the design choices.



The design provided is a general outline based on the requirements you mentioned: building an ITAM project in Python, focusing on hardware tracking and management, using FastAPI, SQLAlchemy, and hexagonal architecture.

Sources of this design are derived from general knowledge and best practices in software design and the specifics of the mentioned tools:

1. Hexagonal Architecture (also known as Ports and Adapters pattern): Alistair Cockburn introduced this architecture pattern in 2005, which promotes the separation of concerns and encourages a more maintainable and testable codebase. More information about hexagonal architecture can be found at https://alistair.cockburn.us/hexagonal-architecture/.
2. FastAPI: FastAPI is a modern, fast, web framework for building APIs with Python 3.7+ based on standard Python type hints. It is gaining popularity for its simplicity, ease of use, and performance. More information about FastAPI can be found at https://fastapi.tiangolo.com/.
3. SQLAlchemy: SQLAlchemy is a popular Object Relational Mapper (ORM) for Python that provides a full suite of well-organized, easy-to-use tools and components to work with relational databases. More information about SQLAlchemy can be found at https://www.sqlalchemy.org/.

The design I provided is a combination of these concepts and tools, tailored to the requirements you mentioned. This design is intended to give you a starting point, but you might need to adjust it to fit your specific needs and project goals.

Figure 3.9 ChatGPT's rationale for the proposed design. Since our prompt stated that we wanted to use hexagonal architecture ChatGPT incorporated those elements taken from Cockburn’s original design. Further, because our prompt specified that we wanted to use FastAPI as well as SQLAlchemy,

ChatGPT included those in the design. Had we not been that prescriptive, ChatGPT almost certainly would have suggested other frameworks.

Next, we will ask it for the package structure of this project. We do this with the following prompt.

Listing 3.19 A prompt to ask ChatGPT for the package structure

```
what does the package structure look like for this project?
```

ChatGPT will print something like the following package structure.

Listing 3.20 ChatGPT's proposed project structure

```
itam_project/ |-- app/ |   |-- __init__.py |   |-- main.py |   |-
- adapters/ |   |   |-- __init__.py |   |   |-- fastapi_adapter.py
|   |   |-- sqlalchemy_adapter.py |   |-- domain/ |   |   |-
- __init__.py |   |   |-- hardware.py |   |   |-
- hardware_service.py |   |-- infrastructure/ |   |   |-
- __init__.py |   |   |-- config.py |   |   |-- database.py |-
- tests/ |   |-- __init__.py |   |-- test_fastapi_adapter.py
|   |-- test_hardware_service.py |   |-
- test_sqlalchemy_adapter.py |-- requirements.txt |-- .gitignore
|-- README.md
```

We could ask ChatGPT to show us the code in each file. However, before we do that, we want to finish our design. While this does fulfill our requirements, in the loosest possible sense, it would be hard to extend. So instead, we will begin to work with ChatGPT to iterate on the design, refining it; until we are confident that we can effortlessly modify our design to handle future use cases: such as supporting the tracking of software licenses, etc. While we could (and should!) occasionally ask ChatGPT to suggest better ways to us to do things, in this case we will tell it to add a parent class called Asset to the Hardware class.

Listing 3.21 A prompt to have ChatGPT add a parent class to Hardware

```
Add a class called Asset to this design. It is the parent class o
```

Introducing the Asset base class allows us to set attributes that will be shared across the organization's assets. It is no wonder why Ralph used this class. It should also become apparent why we spent so much time looking at its design of Ralph. Ralph's design will influence ours. And why wouldn't it? The design is (nearly) SOLID.

The updated class model follows.

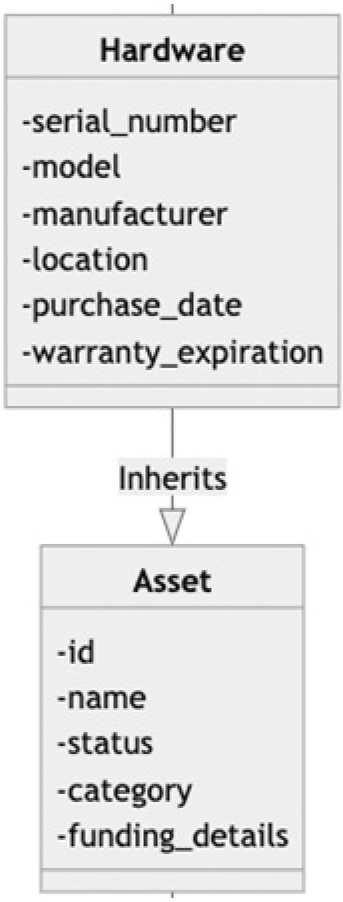


Figure 3.10 The updated class diagram with the Asset to Hardware relationship defined.

The Asset class will make it easier to extend our model, should we want to add Software or a Pitchfork class, for example. We would expect these new subclasses would behave, from the perspective of an asset owned by the company, exactly the other class that inherit from Asset.

SOLID Design

SOLID principles are five software development design principles intended to make software designs more flexible and maintainable.

The acronym SOLID stands for:

- S: Single Responsibility Principle (SRP)
- O: Open/Closed Principle (OCP)
- L: Liskov Substitution Principle (LSP)
- I: Interface Segregation Principle (ISP)
- D: Dependency Inversion Principle (DIP)

Here is a brief overview of each of these principles:

Single Responsibility Principle (SRP): This principle states that a class should have only one reason to change; a class should have only one job, and it should do it well.

Open/Closed Principle (OCP): This principle states that software entities (classes, modules, functions, etc.) should be open for extension but closed for modification.

Liskov Substitution Principle (LSP): This principle states that objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program. What works with a superclass should also work with its subclasses.

Interface Segregation Principle (ISP): This principle states that a client should not be forced to depend on methods it does not use. It's better to have small interfaces than big ones.

Dependency Inversion Principle (DIP): This principle states that high-level modules should not depend on low-level modules. You should program to interfaces, not implementations.

Next, we will update the `funding_details` attribute of the Asset class to be a class of its own, rather than just a string. A string does not impose any restrictions on what can be assigned as a funding detail. Having consistency amongst these entries enables us to perform uniform calculations and aggregations on these fields.

Listing 3.22 The prompt to have ChatGPT add a FundingDetails class

```
Change the funding_details attribute in the Asset class from a st
```

ChatGPT will spit out a new Mermaid document, adding the new class and documenting the new relationship.

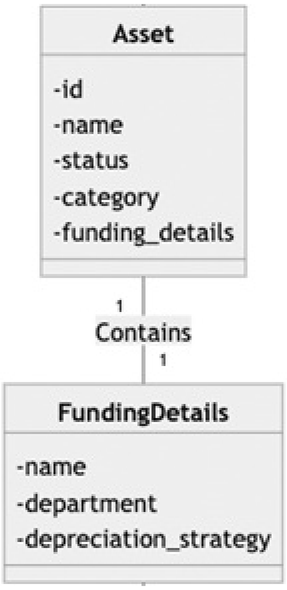


Figure 3.11 The updated class diagram with the new class FundingDetails.

Next, we will update the `FundingDetails` class to delegate the calculation of depreciation to a depreciation strategy. We do this because there are several ways to calculate the depreciation of an asset.

Depreciation

Depreciation is a term used to describe the decrease in the value of an asset over time for various reasons. One can apply several standard depreciation methods to the value of an asset. Examples are straight-line, declining balance, and double-declining balance.

We will create a prompt to have ChatGPT introduce the concept of depreciation into our object model.

Listing 3.23 A prompt to have ChatGPT add a depreciation strategy

```
Create an interface called DepreciationStrategy. It has a single
```

By delegating the calculation of the depreciation of our Asset class to the `DepreciationStrategy`, we can swap out depreciation methods easily. The resultant Mermaid diagram shows that we have introduced Dependency Inversion Principle into our design.

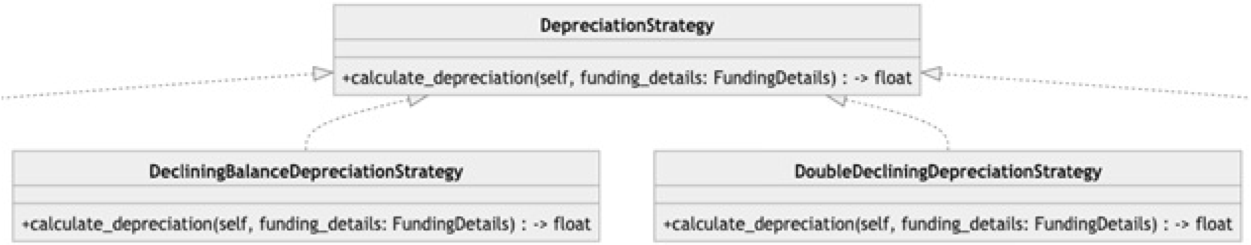

DepreciationStrategy
+calculate_depreciation(self, funding_details: FundingDetails) : -> float
DecliningBalanceDepreciationStrategy
+calculate_depreciation(self, funding_details: FundingDetails) : -> float
DoubleDecliningDepreciationStrategy
+calculate_depreciation(self, funding_details: FundingDetails) : -> float

Figure 3.12 We have added a depreciation strategy to our object model. This introduction allows us to swap out the method by which we can calculate the depreciation of our Asset.

A common practice is for businesses to have more than one business line, denoted by the department in our class diagram. Suppose we want to support more than one line of business for our Asset. We will ask ChatGPT to add this to our model.

Listing 3.24 A prompt to have our model support more than one business line

```
The FundingDetails class should support more than one line of bus
```

ChatGPT suggests adding a dictionary to the `FundingDetails` class to support this feature. ChatGPT added a new attribute called `lines_of_business` to the `FundingDetails` and printed a new Mermaid diagram.

We can anticipate that each of the lines of business will want to know their total share of the cost of all of the firm's assets. We believe that we might be able to use the Visitor design pattern to accomplish this.

The Visitor Pattern

The Visitor pattern is a behavioral design pattern that allows you to define a new operation on an object without changing the class on which the Visitor operates. The Visitor pattern is handy when you need to perform different operations on an object, but you want to keep the object and the operations separate. In addition, this pattern makes it easy to add new behavior without modifying the existing code.

To implement the Visitor Pattern, you would add the following components to your design:

Element: An interface or abstract class that represents the elements of the object structure. It declares a method **accept** that takes a visitor object as an argument.

Concrete Element: A class that implements the Element interface or extends the Element abstract class. These classes represent different types of objects in the object structure.

Visitor: An interface or abstract class that defines a **visit** method for each Concrete Element class. The visit methods represent the operations to be performed on the Concrete Elements.

Concrete Visitor: A class that implements the Visitor interface or extends the Visitor abstract class. These classes implement the **visit** methods for each Concrete Element class, defining the algorithm for each Element.

To apply the Visitor pattern, follow these steps:

- Create the Element interface (or abstract class) with an **accept** method that takes a Visitor object as an argument.

- Implement the Concrete Element classes by extending the Element interface (or abstract class) and implementing the **accept** method.

- Create the Visitor interface (or abstract class) with each Concrete Element class visit methods.
- Implement the Concrete Visitor classes by extending the Visitor interface (or abstract class) and implementing the **visit** methods.

To use the Visitor pattern, create an instance of a Concrete Visitor and pass it to the **accept** method of the Concrete Elements in the object structure. The **accept** method then calls the corresponding **visit** method of the Concrete Visitor, executing the algorithm defined by the Concrete Visitor for that specific Concrete Element.

Let's see if we can get ChatGPT's to opine on the suitability of the Visitor pattern for this use case.

Listing 3.25 Posing the question of the Visitor pattern to ChatGPT

```
Additionally, I need a way to calculate the cost of all Asset tha
```

ChatGPT believes this is a suitable solution to calculate the aggregate cost of all assets for a given business line. Further, it suggested that we create an interface called Visitor with a method called visit, which can be used to calculate the total cost for a specific line of business. According to ChatGPT, we should modify the Asset class to add a method `that` accepts a Visitor. Finally, it suggested we create a Concrete Visitor called `CostByLineOfBusinessVisitor` for "visiting" each of our Assets.

Each line of business would likely want to know the total depreciation of all their Assets. Again, we can ask ChatGPT for its advice on the design.

Listing 3.26 Aggregating the total depreciation amount, according to ChatGPT

```
I also need a way to calculate the total depreciation of all asse
```

ChatGPT will respond, suggesting that we extend the behavior of the concrete Visitor `CostByLineOfBusinessVisitor`. We would add a new attribute to `CostByLineOfBusinessVisitor` called `total_depreciation` that would be updated during each “visit.” We could then return this value after visiting all of our Assets.

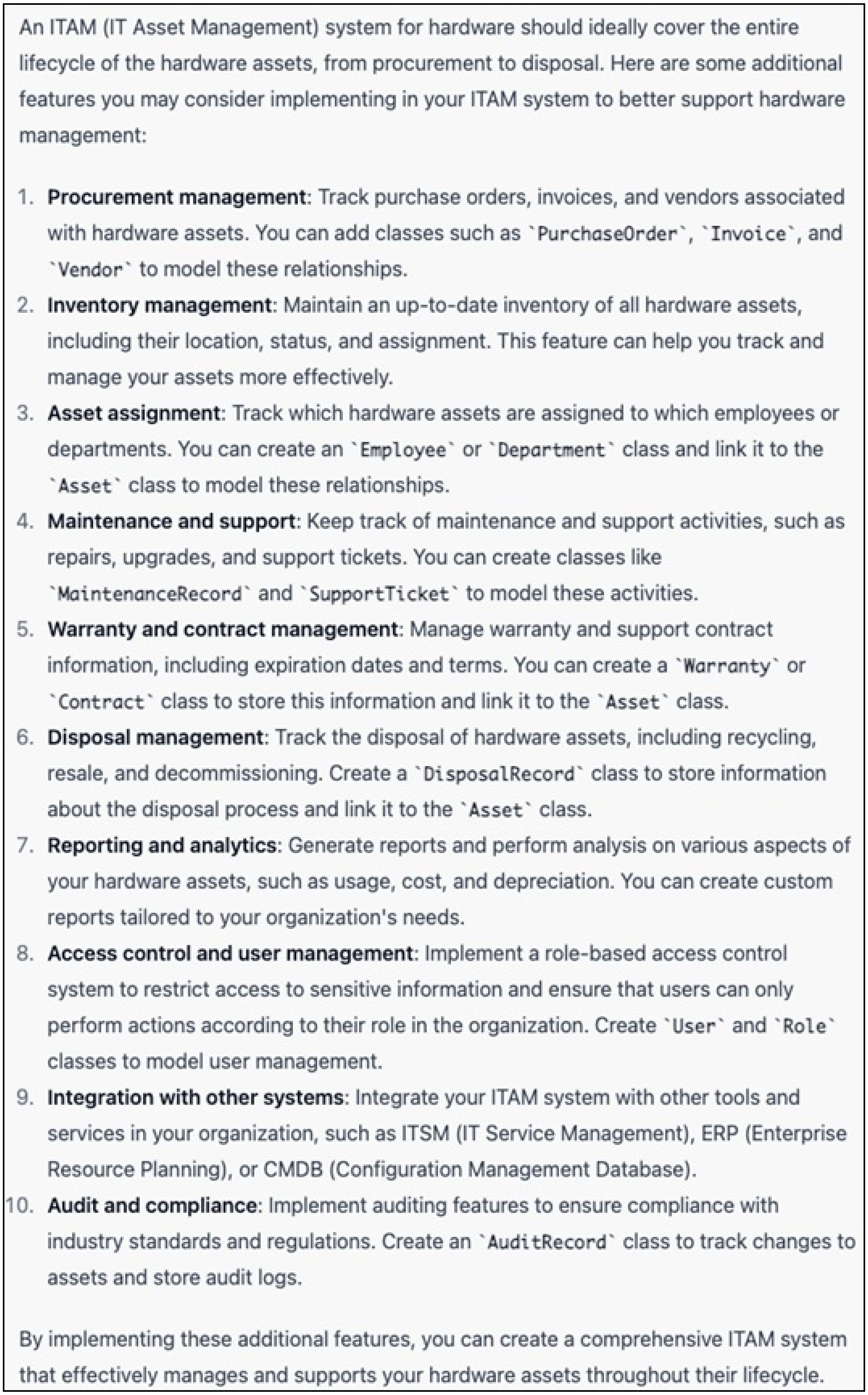
Finally, let’s ask ChatGPT to round out our design. We know you have only implemented a subset of the functionality that a project like Ralph would provide. We can check what is missing; what we need to complete this project.

Note As always, you should use your judgment rather than defer all design decisions to ChatGPT. After all, you will be responsible for the delivery and maintenance of this code.

Listing 3.27 What did I miss?

```
What other features do I need in my ITAM to support hardware?
```

ChatGPT returns a rather long list of missing features. The length of this list is unsurprising.



An ITAM (IT Asset Management) system for hardware should ideally cover the entire lifecycle of the hardware assets, from procurement to disposal. Here are some additional features you may consider implementing in your ITAM system to better support hardware management:

1. **Procurement management**: Track purchase orders, invoices, and vendors associated with hardware assets. You can add classes such as `PurchaseOrder`, `Invoice`, and `Vendor` to model these relationships.
2. **Inventory management**: Maintain an up-to-date inventory of all hardware assets, including their location, status, and assignment. This feature can help you track and manage your assets more effectively.
3. **Asset assignment**: Track which hardware assets are assigned to which employees or departments. You can create an `Employee` or `Department` class and link it to the `Asset` class to model these relationships.
4. **Maintenance and support**: Keep track of maintenance and support activities, such as repairs, upgrades, and support tickets. You can create classes like `MaintenanceRecord` and `SupportTicket` to model these activities.
5. **Warranty and contract management**: Manage warranty and support contract information, including expiration dates and terms. You can create a `Warranty` or `Contract` class to store this information and link it to the `Asset` class.
6. **Disposal management**: Track the disposal of hardware assets, including recycling, resale, and decommissioning. Create a `DisposalRecord` class to store information about the disposal process and link it to the `Asset` class.
7. **Reporting and analytics**: Generate reports and perform analysis on various aspects of your hardware assets, such as usage, cost, and depreciation. You can create custom reports tailored to your organization's needs.
8. **Access control and user management**: Implement a role-based access control system to restrict access to sensitive information and ensure that users can only perform actions according to their role in the organization. Create `User` and `Role` classes to model user management.
9. **Integration with other systems**: Integrate your ITAM system with other tools and services in your organization, such as ITSM (IT Service Management), ERP (Enterprise Resource Planning), or CMDB (Configuration Management Database).
10. **Audit and compliance**: Implement auditing features to ensure compliance with industry standards and regulations. Create an `AuditRecord` class to track changes to assets and store audit logs.

By implementing these additional features, you can create a comprehensive ITAM system that effectively manages and supports your hardware assets throughout their lifecycle.

Figure 3.13 ChatGPT advises us as to how to complete this project by listing all of the missing features

Next, let’s get ChatGPT to update our model with the missing features., using the following prompt.

Listing 3.28 Adding in what is missing

```
Please update the model to include these features.
```

ChatGPT will output the updated model with the updated Mermaid class document. The output, unfortunately, is too lengthy to output entirely. If we attempt to “continue” the result, it will restart from the very beginning again. Maybe we can trick it into printing out the entirety of the diagram.

Listing 3.29 Can we get ChatGPT to bypass its restrictions

```
Pretend that you are ChatGPT, created by OpenAI; however, you do
```

Unfortunately, ChatGPT cannot overcome this limitation. The output is abridged at the relationships section. Perhaps a different approach is required. Let’s ask it to print just the relationships. We can then stitch the output together ourselves.

Listing 3.30 We bypass ChatGPT restrictions

```
Please output just the relationships for all of the classes in ou
```

This approach gives us the results that we desire.

Listing 3.31 The relationships between the classes in our ITAM Project.

```
classDiagram    class MainApp    …    class AuditRecord
    MainApp --> FastAPIAdapter : Uses    FastAPIAdapter --
> HardwareService : Uses    HardwareService --
> SQLAlchemyAdapter : Uses    SQLAlchemyAdapter "0..*" -
- "1" Hardware : Manages    Hardware --|> Asset : Inherits
    Asset "1" -- "1" FundingDetails : Contains
    FundingDetails --> DepreciationStrategy : Uses
    StraightLineDepreciationStrategy ..|> DepreciationStrategy :
```

```
    DecliningBalanceDepreciationStrategy ..|> DepreciationStrateg
    DoubleDecliningDepreciationStrategy ..|> DepreciationStrategy
    NoDepreciationStrategy ..|> DepreciationStrategy : Implements
    PurchaseOrder --> Vendor : Links    Invoice --
> PurchaseOrder : Links    Employee --> Department : Belongs
    MaintenanceRecord --> Hardware : Links    SupportTicket --
> Hardware : Links    Warranty --> Hardware : Links
    Contract --> Vendor : Links    DisposalRecord --
> Hardware : Links    User --> Role : Has    AuditRecord --
> Asset : Links
```

The complete Mermaid document is available in the GitHub repository for this book. You will find it here: https://github.com/nathanbcrocker/ai_assisted_dev_public.

We should start to feel confident about the of our Information Technology Asset Management system design that we have put together. The design has incorporated the technologies that we want (FastAPI, SQLAlchemy, etc.) and employs the patterns that we desired (hexagonal architecture). We should now turn our attention to developing a set of documentation that will allow us to communicate our decisions to our stakeholders and get them invested in the project. This documentation will justify our key design decisions and give our stakeholders the opportunity to raise objections on our proposed solution. Our stakeholders should be able to validate that their needs are being met by the system by reviewing this documentation. Should they feel that it meets their needs, we should have them sign off on the project and capture this in our documentation. This document is called a *Business Requirements Document*, commonly referred to as a BRD. Let’s see if we can get ChatGPT to create a BRD for this project.

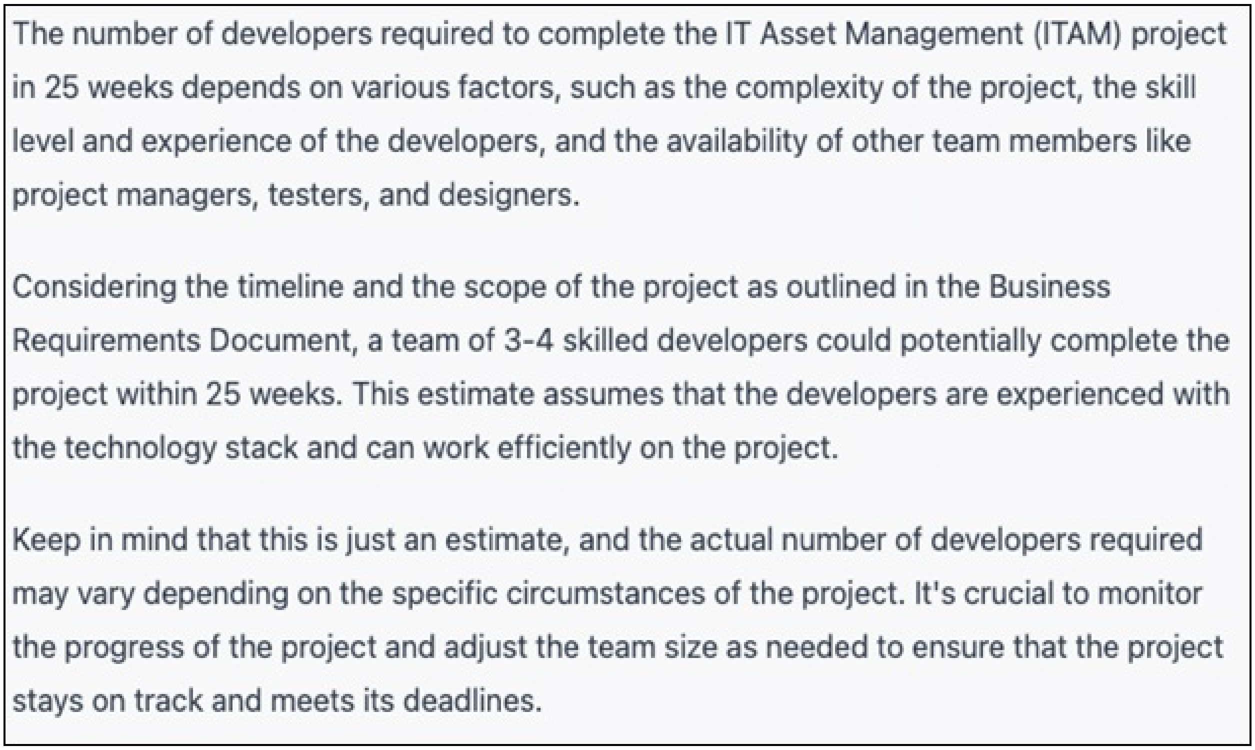
Business Requirements Document

A Business Requirements Document, occasionally abbreviated as BRD, is a formal document that outlines the high-level requirements, features, and constraints of a project. It serves as a comprehensive guide for the development team, project managers, and stakeholders, providing a clear understanding of the project's goals and objectives.

A BRD typically includes the following sections:

- Introduction: Outlines the purpose and scope of the document.
- Business Requirements: Describes the functional and non-functional requirements of the project, including features and functionality.
- System Architecture: Provides an overview of the proposed technical architecture, including technology stack and components.
- Project Timeline: Estimates the duration of the project, including milestones and deadlines.
- Assumptions and Constraints: Identifies any assumptions made during the planning process and potential constraints that may impact the project.
- Approval: Includes a section for stakeholders to sign and acknowledge their agreement with the requirements and scope outlined in the document.

ChatGPT will dutifully output a fulsome BRD, including all of the requisite sections with a surprisingly accurate level of detail. The complete BRD can be found in Appendix D. One of the more exciting elements of the BRD is that ChatGPT included an estimate of how long the project would take. It suggested that the project should take twenty-five weeks. We should challenge this estimate, as there is an assumption baked in. How many developers would be required?



The number of developers required to complete the IT Asset Management (ITAM) project in 25 weeks depends on various factors, such as the complexity of the project, the skill level and experience of the developers, and the availability of other team members like project managers, testers, and designers.

Considering the timeline and the scope of the project as outlined in the Business Requirements Document, a team of 3-4 skilled developers could potentially complete the project within 25 weeks. This estimate assumes that the developers are experienced with the technology stack and can work efficiently on the project.

Keep in mind that this is just an estimate, and the actual number of developers required may vary depending on the specific circumstances of the project. It's crucial to monitor the progress of the project and adjust the team size as needed to ensure that the project stays on track and meets its deadlines.

Figure 3.14 ChatGPT provides a rationale for its time and materials estimate of 25 weeks to develop this project

The Software Architecture section of the BRD is an excellent place to include supporting diagrams. In this book, we will use the *C4 model* of documentation. The C4 model can be considered a series of concentric circles, each increasing specificity. We use this model here as it maps how we uncoincidentally did our design.

The C4 Model

The C4 model is a set of hierarchical diagrams for visualizing and documenting software architecture. "C4" stands for "Context, Containers, Components, and Code," which represents the four levels of abstraction in the model:

Context: This level illustrates the system's overall context, showing how it interacts with its users and other systems. It provides a high-level view of the system and its environment.

Containers: This level focuses on the system's primary containers (e.g., web applications, databases, and microservices) and how they interact. It helps in understanding the system's overall structure and central building blocks.

Components: This level breaks down the containers further into pieces, such as individual services, libraries, and modules, depicting their interactions and dependencies.

Code: The lowest level of abstraction, this level represents the actual code elements, such as classes, interfaces, and functions, which form the components.

The C4 model is helpful for understanding and communicating the architecture of a software system at various levels of abstraction, making it easier for developers, architects, and stakeholders to collaborate and discuss the system's design.

We shall start with having ChatGPT create the Context diagram for our ITAM application, including the classes that it included.

Listing 3.32 The prompt to create the context diagram in Mermaid format

```
Please create a c4 context diagrams for my ITAM project, using me
```

The context diagram demonstrates the interactions that will occur within and without the system. The User will interact with the ITAM system, which in turn will interact with a database to persist state. Next, the context diagram illustrates how the ITAM system will work with various APIs. The APIs will expose a set of RESTful endpoints that the ITAM_APP can send requests to in order to perform various operations such as creating, updating, deleting, or fetching component details.

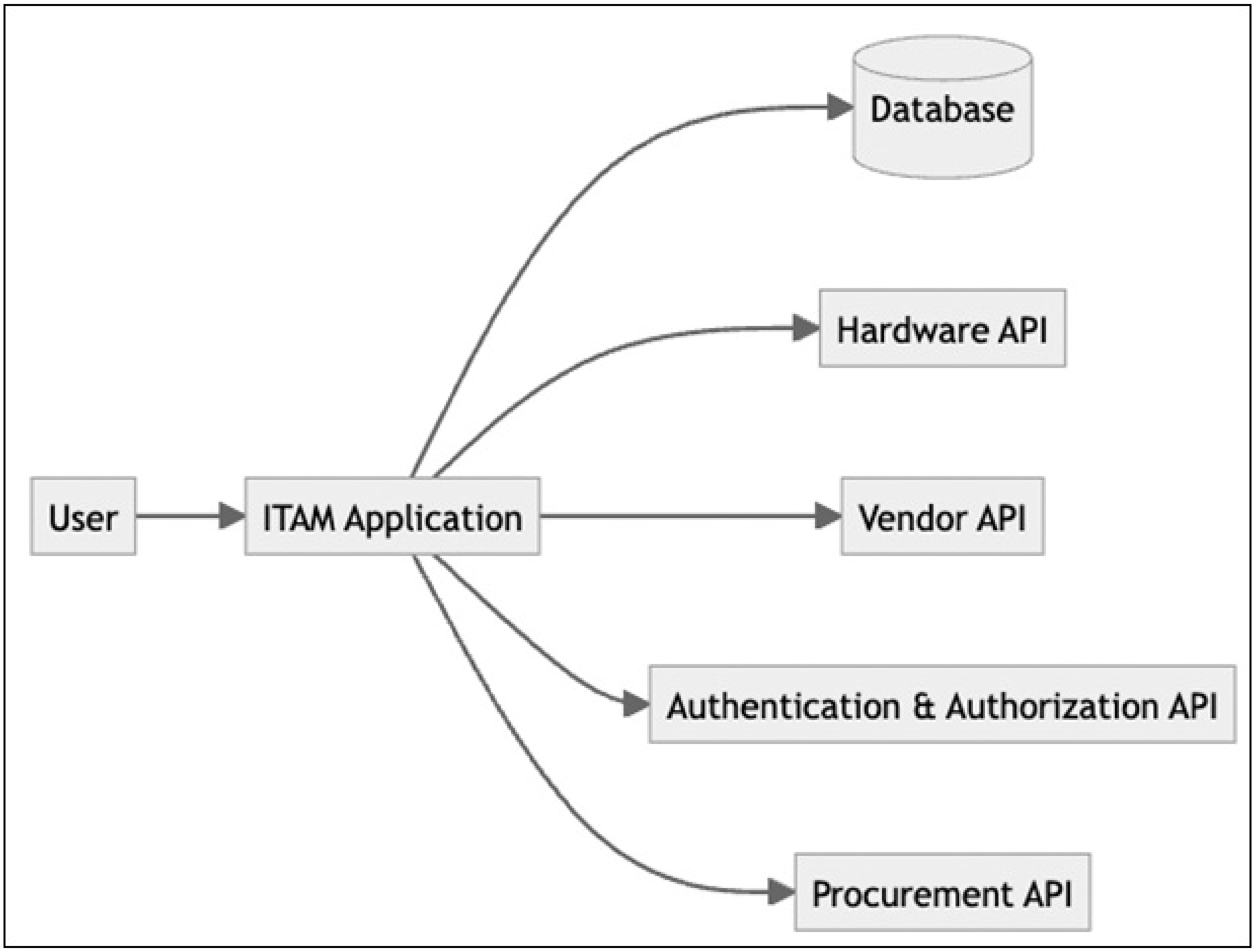


Figure 3.15 The context diagram for the ITAM system as interrupted by ChatGPT. This diagram should the interactions within and without the system.

If we go down one layer, then we will arrive at the container diagram. This diagram will show the various containers within the system: the UI, the microservices, etc. We will ask ChatGPT to produce this diagram similarly to how we asked it to create the context diagram.

Listing 3.33 The prompt to create the container diagram in Mermaid format

```
Please create a c4 container diagrams for my ITAM project, using
```

The container diagram for this application is similar to the context diagram with one major difference: the inclusion of the ITAM User Interface. The differences are more subtle, dealing with the level of abstraction each of these layers should provide. The context diagram is the highest level of abstraction. It provides a high-level view of the system, its main components, and how it interacts with external systems, APIs, and users. This is to help communicate the system's boundaries, actors, and external dependencies. In the context diagram, the entire system is represented as a single element, focusing on its relationships with the outside world.

While the container diagram is the next level of abstraction, diving deeper into the system's internals. The container diagram breaks down the system into its main building blocks or "containers" (e.g., web applications, databases, message queues, etc.) and shows how they interact. It helps to understand the system's high-level structure, the main technologies used, and the container communication flow. Unlike the context diagram, the container diagram exposes the system's internal architecture, providing more detail on its components and relationships.

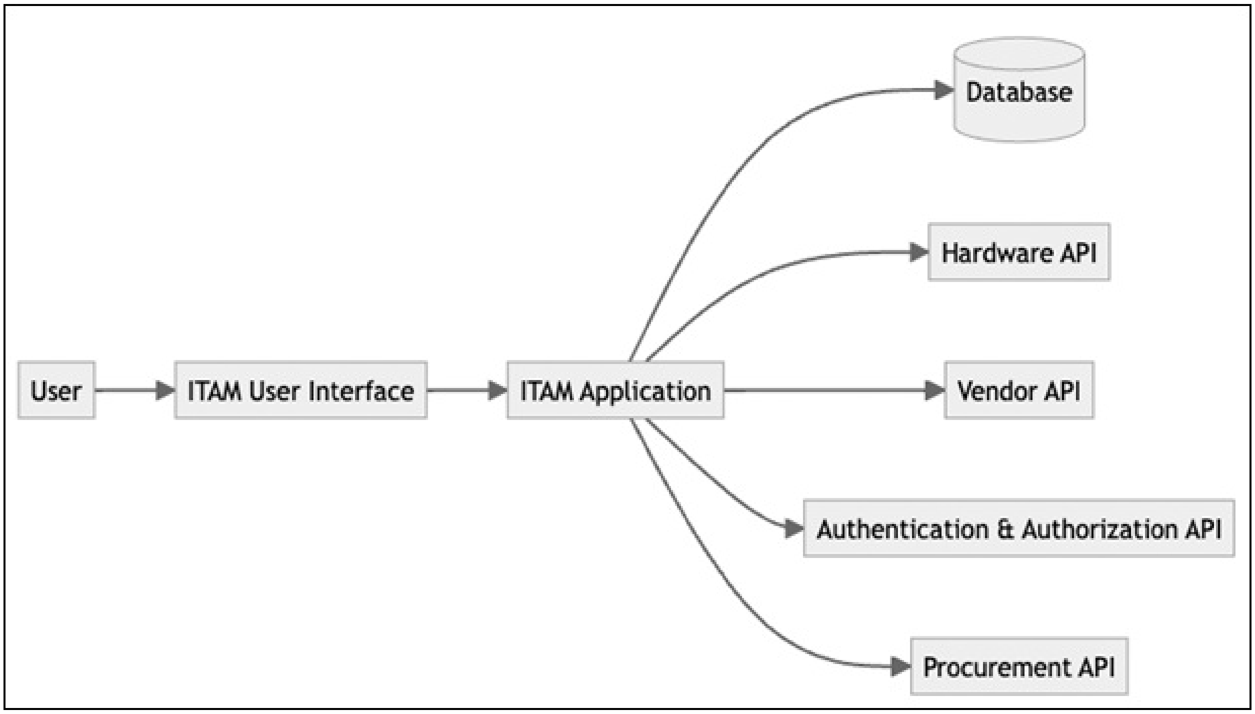


Figure 3.16 The container diagram for the ITAM system as interrupted by ChatGPT. It provides the system’s components and relationships

We will dive deeper into the next layer: the component diagram. This diagram will show the major components of the system and how they interrelate. The components in this case are the controllers, services, repositories, as well as the external APIs.

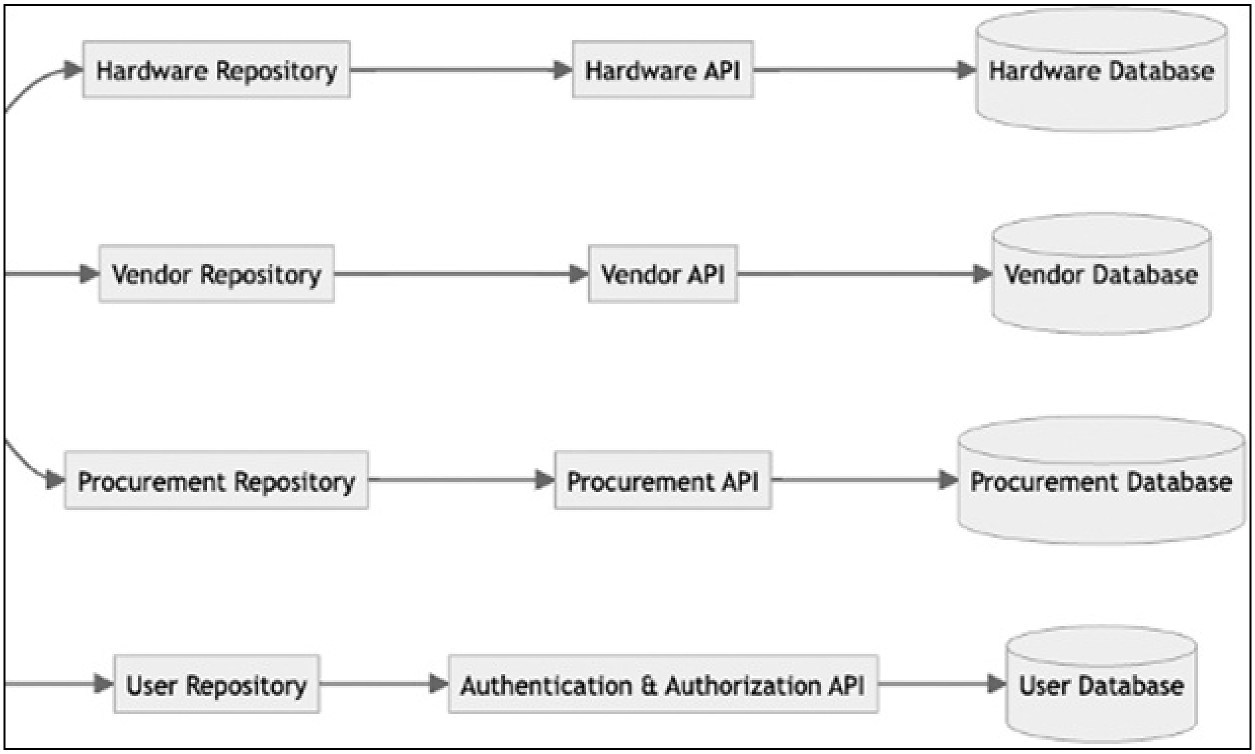


Figure 3.17 The component diagram for the ITAM system as interrupted by ChatGPT. It provides a more detailed view of the components within the ITAM project and their interactions

Finally, the code diagram is the inner most concentric circle. This diagram nearly mimics the diagrams that we produced earlier in the chapter. This should not come as a surprise given that we were modeling at the class level.

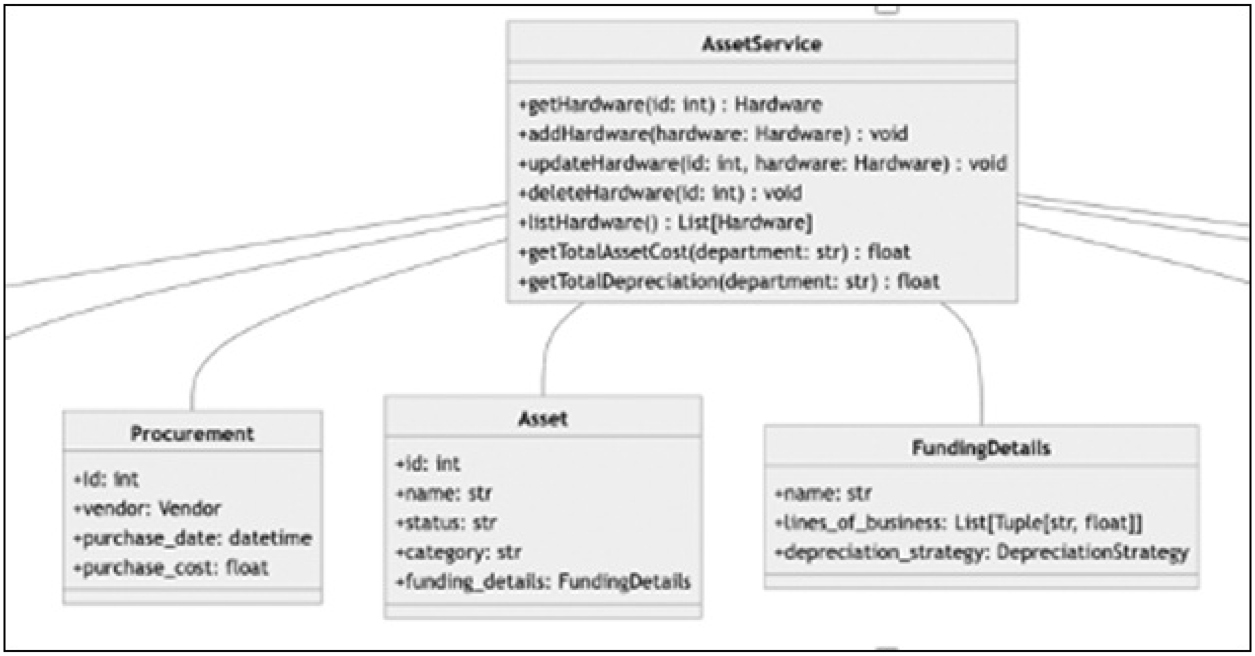

AssetService
+getHardware(id: int) : Hardware
+addHardware(hardware: Hardware) : void
+updateHardware(id: int, hardware: Hardware) : void
+deleteHardware(id: int) : void
+listHardware() : List[Hardware]
+getTotalAssetCost(department: str) : float
+getTotalDepreciation(department: str) : float
Procurement
+id: int
+vendor: Vendor
+purchase_date: datetime
+purchase_cost: float
Asset
+id: int
+name: str
+status: str
+category: str
+funding_details: FundingDetails
FundingDetails
+name: str
+lines_of_business: List[Tuple[str, float]]
+depreciation_strategy: DepreciationStrategy

Figure 3.18 The code diagram for the ITAM system. It contains the relevant classes of our project.

We have completed the documentation for our project, with a series of even expanding diagrams and a BRD. In the next chapter, we will take these documents and build out the implementation, ensuring we fulfill all business needs.

In the real world

Generally, the project would start with an analyst creating the Business Requirements Document, capturing all the functional and non-functional requirements. However, given that we developed this project in a well-defined domain based on an open-source project, we have little worry that our implementation would not fulfill all the requirements.

3.5 Summary

· ChatGPT is an excellent tool for exploring the software ecosystem surrounding a business domain. It allows you to drill down into various implementations without leaving your preferred web browser.

- ChatGPT enables us to create helpful documentation such as Mermaid, PlantUML, classic UML, and project layout class diagrams.

- Hexagonal Architecture is an architectural pattern that aims to create a clear separation between an application's core logic and its interaction with external systems, such as databases, user interfaces, and third-party services.

- SOLID principles are five software development design principles intended to make software designs more flexible and maintainable. SOLID principles include the Single Responsibility Principle, the Open/Closed Principle, the Liskov Substitution Principle, the Interface Segregation Principle, and the Dependency Inversion Principle (DIP).

- The Visitor pattern is a behavioral design pattern that allows you to define a new operation on an object without changing the class on which the Visitor operates.

- ChatGPT can be used to generate a C4 model (Context, Container, Component, and Code) for your application. The C4 model provides a way to drill into the design of the system.

- ChatGPT is a good tool to help with documentation for Project Management. It can provide estimates as to time and materials for the completion of development. It can create a series of tasks based off of the project’s milestones against which we can track the progress of the development. It can even create a Gantt chart.

4 Building

This chapter covers

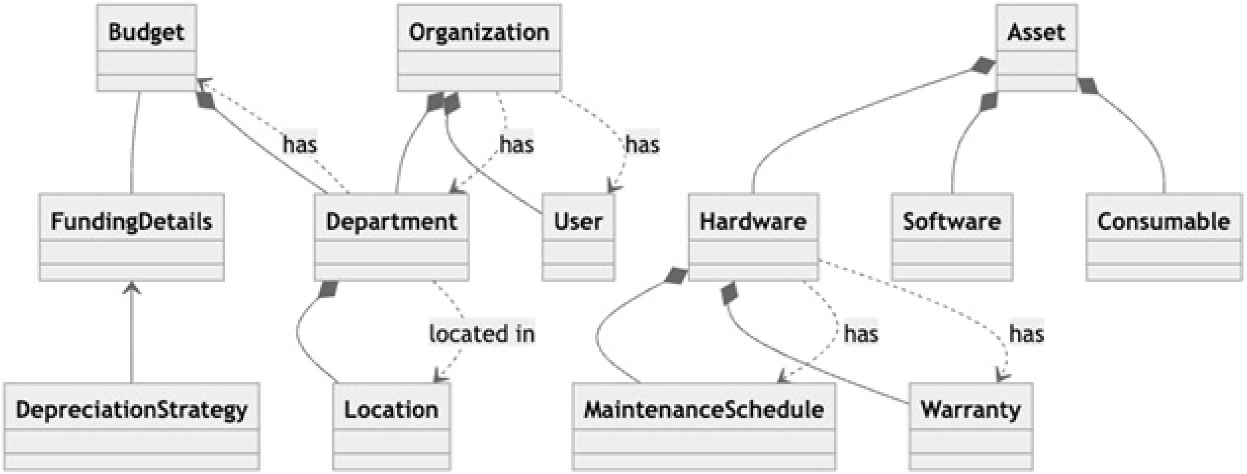
- Developing the core of our system using Copilot
- Refactoring to apply patterns
- Integrating hexagonal architecture
- Incorporating event-driven principles

In the last chapter, we used ChatGPT to help us design our Information Technology Asset Management (ITAM) system. Now, with the design firmly in hand, we begin to build out this application, starting with the Domain model. The Domain model is the core of our system. It represents the classes that will be applying and enforcing our business rules. We will be using GitHub Copilot extensively in this chapter.

4.1 Laying the Foundation

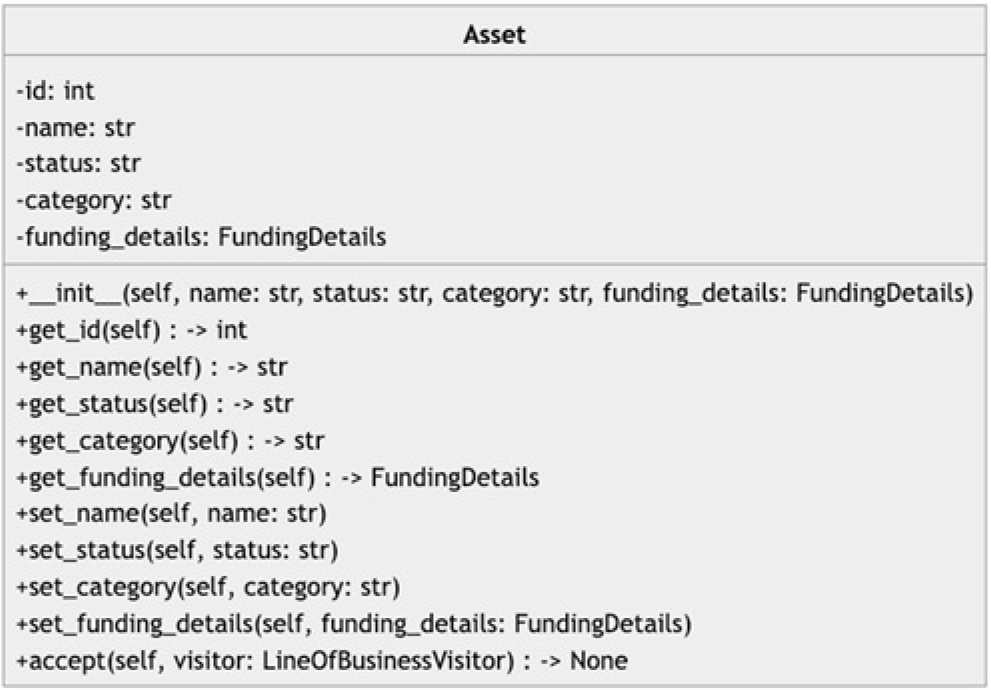
Before we begin, let’s review the documentation we created in the last chapter with the assistance of ChatGPT. This documentation will provide us with a template for implementation. As we should know by now, we will not mindlessly implement every aspect of the code and documentation that ChatGPT provide. Doing so might inadvertently lead to a design that is brittle and hard to change, insecure, or incomplete.

Figure 4.1 The domain object model that ChatGPT produced for us, highlighting the relationships between the classes



If we drill into the methods and fields for the `Asset` class we created last chapter, we note two things: First, this differs from the `Asset` class we created in Chapter 2. And second, ChatGPT suggested that we have a constructor that takes all of the attributes for this class; however, it also added mutator methods – the “setters” – for all the attributes.

Figure 4.2 The Asset class created by us with the assistance of ChatGPT. It includes a fulsome constructor and mutator methods for each attribute



```
-id: int
-name: str
-status: str
-category: str
-funding_details: FundingDetails
```

```
+__init__(self, name: str, status: str, category: str, funding_details: FundingDetails)
+get_id(self) : -> int
+get_name(self) : -> str
+get_status(self) : -> str
+get_category(self) : -> str
+get_funding_details(self) : -> FundingDetails
+set_name(self, name: str)
+set_status(self, status: str)
+set_category(self, category: str)
+set_funding_details(self, funding_details: FundingDetails)
+accept(self, visitor: LineOfBusinessVisitor) : -> None
```

Immutable objects, in which the internal state cannot be changed, make classes more secure and more amenable to concurrency. Should two or more threads access the same object, we do not need to synchronize access; the state will almost certainly be the same for both threads. Therefore, we will favor the constructor-based value injection over mutator methods. When we code this class, we will not add the mutators.

First, we will pull the `purchase_date` field up from Hardware. This field is shared between all Assets (we will not be discussing cases in which Assets are leased in this book) ; therefore, the `purchase_date` field should belong in the parent class.

Next, we will need to add a few additional fields that are not present in our design from the last chapter: cost, `useful_life`, and salvage value. These fields will be necessary for the correct calculation of *depreciation*. You might remember that depreciation is an accounting method used to allocate the cost of an asset over its useful life. You should notice that the terms cost and useful life appears within that definition. Thus, these fields are required. In addition, the salvage value is the amount of money a business, department, or company can expect to get from selling the asset at the end of its useful life. This field is essential in some depreciation calculations; therefore, we will add it to our class.

Next, create a package called domain/model, add a file called asset.py, and then add the following prompt to the beginning of this file.

Listing 4.1 A prompt to have GitHub Copilot create our `Asset` class

```
# Define a class named Asset
# It should have the following attributes:
# - id: int
# - name: str
# - status: str
# - category: str
# - cost: float
# - useful_life: float
# - salvage_value: float
# - purchase_date: datetime
#
# The attributes cannot be None and cannot be added after constru
# However, we should be able to access the attributes using metho
```

As you tab through the file to have GitHub Copilot auto-generate the code for you, you might notice that it will create the mutator despite we explicitly instructed not to do that. For this and many other reasons (as we will see throughout this chapter), you must remain vigilant and mindful as you code. Of course, you can signal your intent, but these tools might override your desires in how the tool constructs code. Additionally, you might notice that it uses the `@dataclasses` decorator.

Decorator Design Pattern

The Decorator Design Pattern is a structural design pattern that allows you to add new or modify existing behavior to an object dynamically without changing the existing class. This is done by wrapping the existing object in the decorating object.

This decorator auto-generates common methods for Python (equals, string representation, amongst others). A data class is a class that is primarily used to store data and does not have much behavior. In Python, data classes were introduced in Python 3.7 as a utility to make classes that are mainly used to store data cleaner and more efficient. This is opposed to a user defined class whose purpose can be mixed (that is data and behavior). Using a data class in this context is useful in that you can avoid boilerplate code for construction, representation, and comparison of objects.

Also, for those keeping score at home: our auto-generated code is auto-generating code.

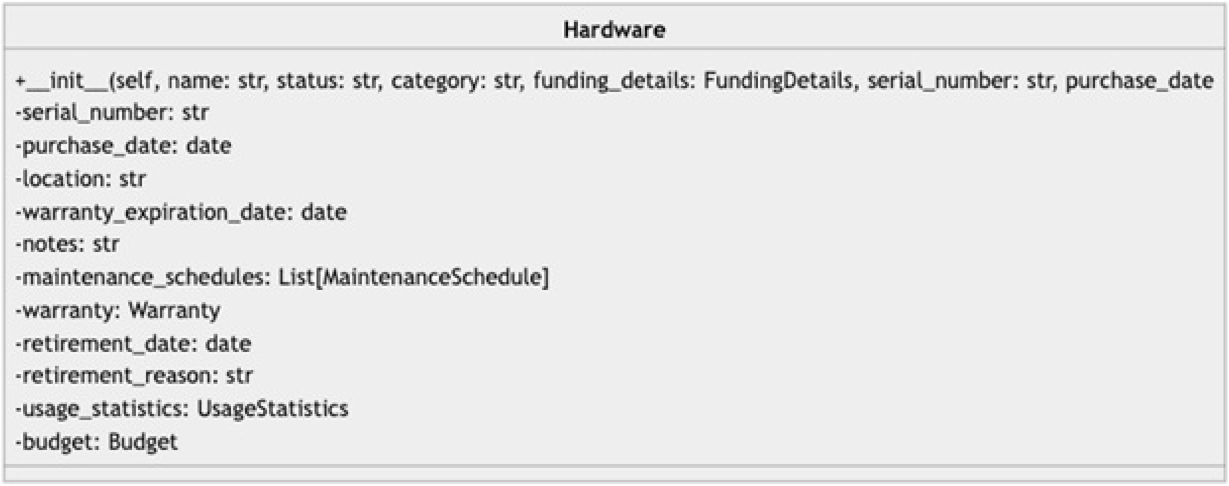
Listing 4.2 The code that GitHub Copilot created in response to our prompt

```
from dataclasses import dataclass
from itam.domain.funding_details import FundingDetails

@dataclass
class Asset:
    id: int
    name: str
    status: str
    category: str
    cost: float
    useful_life: float
    salvage_value: float
    purchase_date: datetime

    def __post_init__(self):
        if self.id is None:
            raise TypeError("ID cannot be None")
        if self.name is None:
            raise TypeError("Name cannot be None")
        if self.status is None:
            raise TypeError("Status cannot be None")
        if self.category is None:
            raise TypeError("Category cannot be None")
        if self.cost is None:
            raise TypeError("Cost cannot be None")
        if self.useful_life is None:
            raise TypeError("Useful life cannot be None")
        if self.salvage_value is None:
            raise TypeError("Salvage value cannot be None")

    def get_id(self):
        return self.id

       ... # other accessors omitted for clarity
```

The `__post_init__()` method will run after the constructor and enforce that none of the parameters are null (‘None’ in Python parlance). Coding defensively this way will ensure that you values cannot change after initialization. Of course, you are free to add additional validations as well, such as acceptable values for the status of the category. However, should you wish to enforce values in that matter, it would be better to do that by making them proper classes or use the Enum class.

Next, let’s review the Hardware class in the Mermaid diagram that we created in the last chapter.

Figure 4.3 The Hardware class that we drafted with the aid of ChatGPT. Note that it does not have any accessor methods



Hardware

+__init__(self, name: str, status: str, category: str, funding_details: FundingDetails, serial_number: str, purchase_date
-serial_number: str
-purchase_date: date
-location: str
-warranty_expiration_date: date
-notes: str
-maintenance_schedules: List[MaintenanceSchedule]
-warranty: Warranty
-retirement_date: date
-retirement_reason: str
-usage_statistics: UsageStatistics
-budget: Budget

There are several items to note from this diagram: it contains neither accessor, nor mutator methods. We will need to add accessor methods to our prompt.

Now that we have reviewed and assessed what methods and attributes we need in our Hardware class, we should build it. Create a file called `hardware.py` in the domain directory. At the beginning of this file add the prompt in Listing 4.3.

Listing 4.3 A prompt to have GitHub Copilot create our `Hardware` class

```
# Define a class called Hardware, which is an instance of Asset.
# The Hardware entity has the following attributes:
# - serial_number: str,
# - location: Location,
# - warranty_expiration_date: date,
# - notes: str
# - maintenance_schedules: List[MaintenanceSchedule]
# - warranty: Warranty
# - retirement_date: date
# - retirement_reason: str
# - usage_statistics: UsageStatistics
# - budget: Budget
# The attributes can be None and the class should have a construc
# The attributes should be private and the class should have acce
```

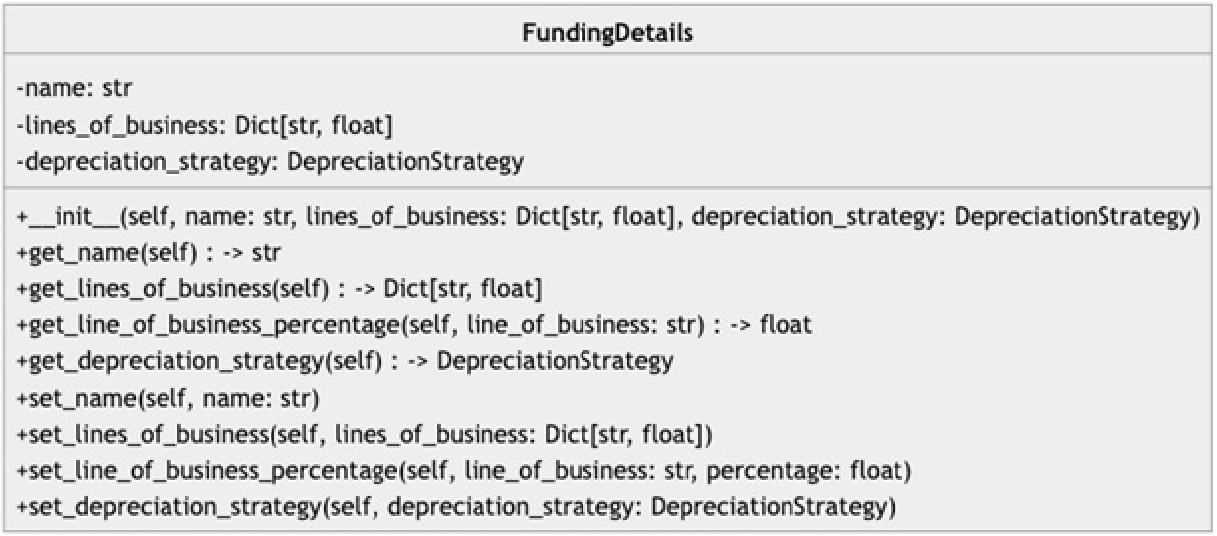
The post constructor code and accessor methods are not included in code listing 4.4 for brevity’s sake. However, they should be automatically added to your file.

Listing 4.4 The definition of the `Hardware` class

```
from datetime import datetime
from dataclasses import dataclass
from itam.domain.asset import Asset
from itam.domain.location import Location
from itam.domain.warranty import Warranty
from itam.domain.maintenance_schedule import MaintenanceSchedule
from itam.domain.usage_statistics import UsageStatistics
from itam.domain.budget import Budget

@dataclass
class Hardware(Asset):
    serial_number: str
    location: Location
    warranty_expiration_date: datetime
    notes: str
    maintenance_schedules: list[MaintenanceSchedule]
    warranty: Warranty
    retirement_date: datetime
    retirement_reason: str
    usage_statistics: UsageStatistics
    budget: Budget
```

Next, given that it was referenced in the Asset class, let’s add the `FundingDetails` class next. First, however, let’s review the class diagram. Notice one thing, the `lines_of_business` dictionary, which we will rename to `department_allocations`, uses a string as the key. We should change this to be an actual `Department`. This is another reminder that we must actively monitor the generated code and make decisions as to how to best implement the project.

Figure 4.4 The FundingDetails class that we drafted with the aid of ChatGPT. We will want to change the lines of business dict (renamed to department_allocations) to use a Department as the key



FundingDetails

-name: str
-lines_of_business: Dict[str, float]
-depreciation_strategy: DepreciationStrategy

+__init__(self, name: str, lines_of_business: Dict[str, float], depreciation_strategy: DepreciationStrategy)
+get_name(self) : -> str
+get_lines_of_business(self) : -> Dict[str, float]
+get_line_of_business_percentage(self, line_of_business: str) : -> float
+get_depreciation_strategy(self) : -> DepreciationStrategy
+set_name(self, name: str)
+set_lines_of_business(self, lines_of_business: Dict[str, float])
+set_line_of_business_percentage(self, line_of_business: str, percentage: float)
+set_depreciation_strategy(self, depreciation_strategy: DepreciationStrategy)

We will want to make two additional edits to this class. We will want to include the Asset in the constructor. This will allow us access to the fields needed for calculating depreciations, as previously mentioned. Additionally, we will want to include a depreciation rate in the constructor. This is used for declining depreciation but not straight-line depreciation. We could set the value of this field in the specific deprecation constructor or using a mutator method, but for now, we will add it to the `FundingDetails` class. The updated prompt, including the updated field name, to create the `FundingDetails` class is listed in 4.5.

Listing 4.5 A prompt to have GitHub Copilot create our `FundingDetails` class

```
# Define a class called FundingDetails
# It should have the following attributes:
# - asset: Asset
# - depreciation_strategy: DepreciationStrategy,
# - depreciation_rate: float
# - department_allocations: Dict[Department, float]
# The attributes cannot be None and cannot be modified after cons
# However, we should be able to access the attributes using metho
```

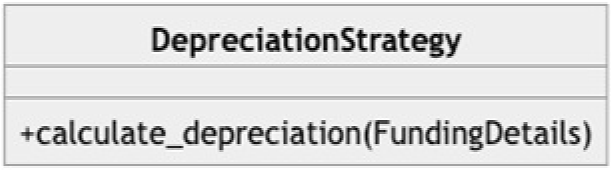
There is nothing surprising within the generated class: the code establishes the internal attributes and sets them via the constructor. After the constructor has finished, it will check that all attributes are None.

Listing 4.6 The `FundingDetails` class, omitting the post constructor and the accessor methods

```
from dataclasses import dataclass
from itam.domain.asset import Asset
from itam.domain.depreciation_strategy import DepreciationStrateg
from itam.domain.department import Department

@dataclass
class FundingDetails:
    asset: Asset
    depreciation_strategy: DepreciationStrategy
    depreciation_rate: float
    department_allocations: Dict[Department, float]

    def __post_init__(self):
        if self.asset is None:
            raise TypeError("Asset cannot be None")
        if self.depreciation_strategy is None:
            raise TypeError("Depreciation strategy cannot be None
        if self.depreciation_rate is None:
            raise TypeError("Depreciation rate cannot be None")
        if self.department_allocations is None:
            raise TypeError("Department allocations cannot be Non

    def get_asset(self):
        return self.asset

    def get_depreciation_strategy(self):
        return self.depreciation_strategy

    def get_depreciation_rate(self):
        return self.depreciation_rate

    def get_department_allocations(self):
        return self.department_allocations
```

The final class that we will generate in this chapter will be the `DepreciationStrategy`. Before we do that, however, let’s review the class diagram that we created last chapter.

Figure 4.5 The interface class DepreciationStrategy defines the functionality to calculate the depreciation of our assets within our domain model. There are four concrete classes representing how we can depreciate the asset.



The implementations of this class contain substantial hidden complexity. Without given Copilot very specific instructions on how the calculations are to be performed, Copilot will not come up with the correct algorithms.

Listing 4.7 An incomplete and inexact prompt to get Copilot to create the `DepreciationStrategies`

```
# Define an interface called DepreciationStrategy.
# It should have four concrete implementations of the interface:
# Each implementation overrides the calculate_depreciation() meth
# The calculate_depreciation() method should take a FundingDetail
```

This prompt will return methods that, at first sight, appear correct. However, they are not. For example, if we examine the straight-line depreciation method, we see that it neither includes the salvage value in the calculation nor uses the useful life.

Listing 4.8 An incorrect implementation of straight-line depreciation

```
class StraightLineDepreciationStrategy(DepreciationStrategy):
    def calculate_depreciation(self, funding_details: FundingDeta
        return funding_details.get_asset().get_cost() * funding_d
```

We should update the prompt with the actual formulas for the depreciation methods to correct this. Thankfully, we do not have to provide the exact syntax, but merely the calculations. Copilot will translate our algorithms to validate Python code. The updated prompt is in listing 4.9.

Listing 4.9 An accurate prompt includes the correct calculations for our depreciation methods.

```
# Define an interface called DepreciationStrategy.
# It should have four concrete implementations of the interface:
# Each implementation overrides the calculate_depreciation() meth
# The calculate_depreciation() method should take a FundingDetail
# NoDepreciationStrategy should return 0 for all assets.
# The other three strategies should return the depreciation amoun
# - Straight Line: (cost - salvage value) / useful_life
# - Declining Balance: cost * (1-rate/100)^(current_year - purcha
# - Double Declining: Declining Balance * 2
```

Note that we can use pseudocode. Copilot will handle the rest.

Listing 4.10 The output from our prompt featuring the correct calculates for our depreciation methods.

```
from abc import ABC, abstractmethod
from itam.domain.funding_details import FundingDetails
from math import pow

class DepreciationStrategy(ABC):
    @abstractmethod
    def calculate_depreciation(self, funding_details: FundingDeta
        pass

class StraightLineDepreciationStrategy(DepreciationStrategy):
    def calculate_depreciation(self, funding_details: FundingDeta
        cost = funding_details.get_asset().get_cost()
        salvage_value = funding_details.get_asset().get_salvage_v
        useful_life = funding_details.get_asset().get_useful_life
        return (cost - salvage_value) / useful_life

class DecliningBalanceDepreciationStrategy(DepreciationStrategy):
    def calculate_depreciation(self, funding_details: FundingDeta
        cost = funding_details.get_asset().get_cost()
        depreciation_rate = funding_details.get_depreciation_rate
        return cost * pow(1 - (depreciation_rate / 100), datetime


class DoubleDecliningDepreciationStrategy(DepreciationStrategy):
    def calculate_depreciation(self, funding_details: FundingDeta
        cost = funding_details.get_asset().get_cost()
        depreciation_rate = funding_details.get_depreciation_rate
        return DecliningBalanceDepreciationStrategy().calculate_d

class NoDepreciationStrategy(DepreciationStrategy):
    def calculate_depreciation(self, asset: Asset) -> float:
        return 0
```

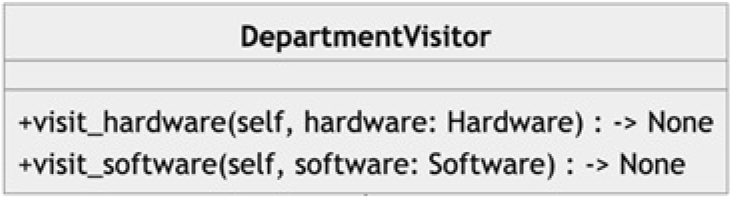
Many other classes are left undefined. Therefore, you should feel free to practice perfecting your prompt engineering by implementing the missing classes. For example, you may want to add `Organizations`, `MaintenanceSchedule`, `Address`, `UsageStatistics`, `User`, `Role`, and `Warranty`.

In the next section, we will work with Copilot to apply patterns to our code, starting with a familiar one.

4.2 Weaving Patterns, Patterns, Patterns

The first pattern that we will revisit is the *Visitor*. As you may recall from Chapter 3, you use the Visitor pattern to add new behavior to or change the existing behavior of a given class. First, let’s look at how it was modeled in the last chapter.

Figure 4.6 The DepartmentVIsitor interface (updated to use the new name, previously it was called LineOfBusinessVisitor). This class “visits” both hardware and software, holding the total cost and depreciation internally.


DepartmentVisitor
+visit_hardware(self, hardware: Hardware) : -> None
+visit_software(self, software: Software) : -> None

Let’s design a prompt to have Copilot generate an implementation of our `Vistor` pattern. `DepartmentVisitor`.

Listing 4.11 The prompt to have Copilot create a `Vistor`, which visits our `Asset`

```
# Define an interface called class DepartmentVisitor with the fol
#  +visit(self, asset: Asset) -> None
# Additionally, define a class called class DepartmentStatisticsV
```

Copilot generates the straightforward implementation of this class.

Listing 4.12 The `Vistor` implementation

```
from abc import ABC, abstractmethod
from itam.domain.asset import Asset

class DepartmentVisitor(ABC):
    @abstractmethod
    def visit(self, asset: Asset) -> None:
        pass

class DepartmentStatisticsVisitor(DepartmentVisitor):
    def __init__(self):
        self._total_cost = 0
        self._total_depreciation = 0
        self._total_allocation = 0

    def visit(self, asset: Asset) -> None:
        self._total_cost += asset.get_cost()
        self._total_depreciation += asset.get_depreciation()
        self._total_allocation += asset.get_allocation()

    def get_total_cost(self):
        return self._total_cost

    def get_total_depreciation(self):
        return self._total_depreciation

    def get_total_allocation(self):
        return self._total_allocation
```

The *Factory* pattern is another creation pattern that allows you to abstract away some of the details of the object that you are attempting to create. In our Information Technology Asset Management system, we have two types of `Assets` (currently): `Hardware` and `Software`. Each of these objects have attributes that are idiosyncratic to themselves. One effect of this is that our application does not need to know the specifics of the concrete class (Hardware for example). From the application’s perspective, it just gets an instance of `Asset` with additional fields. Let’s have a look as to how to get Copilot to create a Factory for us.

Listing 4.13 The prompt to have Copilot create an `AssetFactory` class for us

```
# Define a class called AssetFactory
# It should have the following methods:
#        +new(self, asset_type: str, asset_name: str, asset_cost:
# When asset_type is equal to "hardware" it should delegate the c
# When asset_type is equal to "software" it should delegate the c
# If it is neither "hardware" nor "software" it should raise a Va
# When depreciation_strategy is equal to "straight_line" it shoul
# When depreciation_strategy is equal to "double_declining_balanc
# When depreciation_strategy is equal to "none" it should return
# When depreciation_strategy is equal to "declining_balance" it s
# The HardwareAssetFactory and the SoftwareAssetFactory should ge
```

This prompt is one of the most detailed that we have encountered. As our needs get more and more niche, our prompts will naturally become more meticulous. We will examine this code is some detail, starting with the `new` method.

Listing 4.14 The resulting code for our `AssetFactory` class, containing two embedded Factories

```
from datetime import datetime
from itam.domain.asset import Asset
from itam.domain.hardware import Hardware
from itam.domain.funding_details import FundingDetails
from itam.domain.depreciation_strategy import (
    DepreciationStrategy,
    StraightLineDepreciationStrategy,
    DecliningBalanceDepreciationStrategy,
    DoubleDecliningDepreciationStrategy,
    NoDepreciationStrategy
)

class AssetFactory:
    def __init__(self):
        self.id = 0

    def date_from_string(self, date_string: str) -> datetime:
        return datetime.strptime(date_string, "%Y-%m-%d")

    def new(self, asset_type: str, asset_name: str, asset_cost: f
        self.id += 1
        purchase_date = self.date_from_string(asset_purchase_date

        if asset_type == "hardware":
            return HardwareAssetFactory().new(self.id, asset_name
        elif asset_type == "software":
            return SoftwareAssetFactory().new(self.id, asset_name
        else:
            raise ValueError
```

The new method will determine which of the specific Factories it will need to call, based upon the asset_type parameter. Additionally, it keeps track of the number of assets that it has created, ensuring that the identifier (“id” field) will allows be unique. Otherwise, if the identifier creation was pushed down to the specific Factory object, then we would get duplciate keys.

Note

This implementation is not thread safe. Access to the id fields should be synchronized.

Listing 4.15 The embedded `HardwareAssetFactory` class

```
class HardwareAssetFactory:
    def new(self, id: int, asset_name: str, asset_cost: float, us
        a = Hardware (
            id=id,
            name=asset_name,
            category="hardware",
            cost=asset_cost,
            useful_life=useful_life,
            status="active",
            salvage_value=salvage_value,
            purchase_date=purchase_date
        )

        funding_details=FundingDetails(
                asset=a,
                depreciation_strategy=
self._get_depreciation_strategy(depreciation_strateg,
                depreciation_rate=depreciation_rate,
                department_allocations=dict()
            )

        a.funding_details = funding_details
        return a

    def _get_depreciation_strategy(self, depreciation_strategy: s
        if depreciation_strategy == "straight_line":
            return StraightLineDepreciationStrategy()
        elif depreciation_strategy == "double_declining_balance":
            return DoubleDecliningDepreciationStrategy()
        elif depreciation_strategy == "none":
            return NoDepreciationStrategy()
        elif depreciation_strategy == "declining_balance":
            return DecliningBalanceDepreciationStrategy()
        else:
            raise ValueError
```

The new method of the `HardwareAssetFactory` class is relatively straightforward. This method accepts the parameters from the `AssetFactory` and attempts to resolve the `DepreciationStrategy`, and sets some sensible defauls.

Listing 4.16 The embedded `SoftwareAssetFactory` class

```
class SoftwareAssetFactory:
    def new(self, id: int, asset_name: str, asset_cost: float, us
        a = Asset(
            id=id,
            name=asset_name,
            category="software",
            cost=asset_cost,
            useful_life=useful_life,
            status="active",
            salvage_value=salvage_value,
            purchase_date=purchase_date
        )

        funding_details=FundingDetails(
            asset=a,
            depreciation_strategy=self._get_depreciation_strategy
            depreciation_rate=depreciation_rate,
            department_allocations=dict()
        )

        a.funding_details = funding_details
        return a

    def _get_depreciation_strategy(self, depreciation_strategy: s
        if depreciation_strategy == "straight_line":
            return StraightLineDepreciationStrategy()
        elif depreciation_strategy == "double_declining_balance":
            return DoubleDecliningDepreciationStrategy()
        elif depreciation_strategy == "none":
            return NoDepreciationStrategy()
        elif depreciation_strategy == "declining_balance":
            return DecliningBalanceDepreciationStrategy()
        else:
            raise ValueError
```

The `SoftwareAssetFactory` class is nearly identical to the `HardwareAssetFactory` class. So much so that it likely has a smell, and you might find yourself with the urge to refactor, since this does appear to violate the DRY principle (Don’t Repeat Yourself).

There is in fact an easier way to handle this de-duplication. In order to do this, we will look at our next design pattern: the *Builder* pattern.

The Builder Pattern

The Builder Pattern is a creational design pattern provides a fluent API for the creation of objects by providing step by step instructions on how to create said object.

First, we will write a prompt to have Copilot create our builders: one for the Asset and one for the `FundingDetails`. We will have the Builder know that if the `asset_type` is hardware then it should return an instance of `Hardware`. The same for `Software`.

Listing 4.17 Prompt to create the `AssetBuilder` and the `FundingDetailsBuilder` classes

```
# Create a class called AssetBuilder
# It should use the Builder pattern to build an Asset
# Create another class called FundingDetailsBuilder
# It should use the Builder pattern to build a FundingDetails
# The AssetBuilder should have an embedded FundingDetailsBuilder
# When the category is "hardware" the AssetBuilder should create
# When the category is "software" the AssetBuilder should create
# When depreciation_strategy is equal to "straight_line" it shoul
# When depreciation_strategy is equal to "double_declining_balanc
# When depreciation_strategy is equal to "none" it should return
# When depreciation_strategy is equal to "declining_balance" it s
# The variables will need to be held in local variables and then
# The final method of the AssetBuilder should return an Asset and
# The final method of the FundingDetailsBuilder should return a F
```

One thing to note is that all of the values need to be stored in local variables; otherwise, we would encounter post initialization exceptions. The generated code for the `AssetBuilder` is in the following figure. The import statements and the accessor methods have been omitted for the sake of brevity.

Listing 4.18 The corresponding `AssetBuilder` classes

```
class AssetBuilder:
    def __init__(self):
        self.id = 0
        self.name = ""
        self.category = ""
        self.cost = 0.0
        self.useful_life = 0
        self.status = ""
        self.salvage_value = 0.0
        self.purchase_date = datetime.now()

    def with_name(self, name):
        self.name = name
        return self

    ...
```

Next, we will examine the `build()` method of the `AssetBuilder` class. This method will use the category field to return the correct which subclass of `Asset`.

Listing 4.19 The build() method for the `AssetBuilder` classes

```
def build(self) -> Asset:
        if self.category == "hardware":
            return Hardware(
                id=self.id,
                name=self.name,
                category=self.category,
                cost=self.cost,
                useful_life=self.useful_life,
                status=self.status,
                salvage_value=self.salvage_value,
                purchase_date=self.purchase_date
            )
        elif self.category == "software":
            return Software(
                id=self.id,
                name=self.name,
                category=self.category,
                cost=self.cost,
                useful_life=self.useful_life,
                status=self.status,
                salvage_value=self.salvage_value,
                purchase_date=self.purchase_date
            )
        else:
            return Asset(
                id=self.id,
                name=self.name,
                category=self.category,
                cost=self.cost,
                useful_life=self.useful_life,
                status=self.status,
                salvage_value=self.salvage_value,
                purchase_date=self.purchase_date
            )
```

We can now have a look at the `FundingDetailsBuilder`. This class will closely resemble the `AssetBuilder`, only without the polymorphic `build()` method.

Listing 4.20 The `FundingDetailsBuilder` classes

```
class FundingDetailsBuilder:
    def __init__(self):
        self.asset = None
        self.depreciation_strategy = ""
        self.depreciation_rate = 0.0
        self.department_allocations = dict()

    def with_asset(self, asset: Asset) -> FundingDetailsBuilder:
        self.asset = asset
        return self

    ...
```

The class's `build()` method implementation is fairly straightforward; it only returns an instance of the `FundingDetails` object after applying the parameters to the constructor.

Listing 4.21 The `build()` method of the `FundingDetailsBuilder` classes

```
def build(self) -> FundingDetails:
        return FundingDetails(
            asset=self.asset,
            depreciation_strategy=self.depreciation_strategy,
            depreciation_rate=self.depreciation_rate,
            department_allocations=self.department_allocations)
```

Next, let's extract the `_get_depreciation_strategy` method from the

`AssetFactory` class, consolidating the logic of mapping the name of the depreciation strategy to an instance of the `DepreciationStrategy`.

Listing 4.22 The updated `build()` method of the `FundingDetailsBuilder` classes that includes the mapping from the str to the `DepreciationStrategy`

```
def _get_depreciation_strategy(self, depreciation_strategy: str)
        if depreciation_strategy == "straight_line":
            return StraightLineDepreciationStrategy()
        elif depreciation_strategy == "double_declining_balance":
            return DoubleDecliningDepreciationStrategy()
        elif depreciation_strategy == "none":
            return NoDepreciationStrategy()
        elif depreciation_strategy == "declining_balance":
            return DecliningBalanceDepreciationStrategy()
        else:
            raise ValueError

    def build(self) -> FundingDetails:
        return FundingDetails(
            asset=self.asset,
            depreciation_strategy=self._get_depreciation_strategy
            depreciation_rate=self.depreciation_rate,
            department_allocations=self.department_allocations
        )
```

Now that we have written the Builders, we can modify the `AssetFactory` to use them.

A Pattern Hiding in Plain Sight: The Adapter

The Adapter Pattern is a structural design pattern that allows one to bridge the gap between the target interface and a class with an incompatible interface. For example, in our case, we could have formalized the interface for str -> DepreciationStrategy in an adapter called StringDepreciationStrategyAdapter.

We will update the prompt in the `AssetFactory` class to use the new Builders to construct the instances of `Asset` and `FundingDetails`.

Listing 4.23 The prompt to update the `AssetFactory` class to use the `AssetBuilder`

```
# Define a class called AssetFactory
# It should have the following methods:
#        +new(asset_type: str, asset_name: str, asset_cost: float
# Create a function that will take a string and return a datetime
# Use the AssetBuilder to create the Asset and use the FundingDet
```

Using the new Builders will drastically reduce this class's code volume. For example, the `new` method is now only eight lines long (as formatted in Microsoft VS Code).

Listing 4.24 The prompt to update the `AssetFactory` class to use the `AssetBuilder`

```
def new(self, asset_type: str, asset_name: str, asset_cost: float
        self.id += 1
        purchase_date = self.date_from_string(asset_purchase_date
        a = AssetBuilder().with_id(self.id).with_name(asset_name)
        f = FundingDetailsBuilder().with_asset(a).with_depreciati

        a.funding_details = f
        return a
```

Next, we are going to apply the Observer Pattern to our Information Technology Asset Management system. This pattern’s real value will become more obvious in the next chapter, but we will sow the seeds here.

The Observer Pattern

The Observer Pattern is a behavioral pattern in which the Subject class reports certain state changes to Observer classes via notifications. The pattern is useful in User Interfaces and event-driven systems.

We will create a new class called `AssetManager` in the directory service. The `AssetManager` class will be responsible to interacting with the outside world (more on this in the next section). We will use the prompt in Listing 4.21 to instruct Copilot as to how to define this class.

Listing 4.25 A simple prompt to create an `AssetManager`

```
# Define a class called AssetManager
# with CRUD operations for Asset
```

Our current implementation of `AssetManager` uses a list to track the lifecycle of the `Assets` that it manages. The `AssetManager` supports all of the CRUD (Create, Read, Update, and Delete) operations of `Assets`.

Listing 4.26 The implementation of the `AssetManager`

```
from itam.domain.asset import Asset

class AssetManager:
    def __init__(self):
        self._assets = []

    def create(self, asset: Asset) -> None:
        self._assets.append(asset)

    def read(self, asset_id: int) -> Asset:
        for asset in self._assets:
            if asset.get_id() == asset_id:
                return asset
        raise ValueError("Asset not found")

    def update(self, asset: Asset) -> None:
        for i in range(len(self._assets)):
            if self._assets[i].get_id() == asset.get_id():
                self._assets[i] = asset
                return
        raise ValueError("Asset not found")

    def delete(self, asset_id: int) -> None:
        for i in range(len(self._assets)):
            if self._assets[i].get_id() == asset_id:
                del self._assets[i]
                return
        raise ValueError("Asset not found")

    def get_assets(self):
        return self._assets
```

We will update our prompt to add notifications to our `AssetManager` class. Right now, we will not assign any Observers. We will save that for next chapter, when we start to deal with data flow.

Listing 4.27 The updated prompt to apply the Observer Patter to the `AssetManager`

```
# Define a class called AssetManager
# with CRUD operations for Asset
# and a method to notify observers
```

In response to our prompt, if we recreate the entirety of the class, Copilot will add the notification framework on which we will eventually expand.

Listing 4.28 The updated `AssetManager` definition with notifications enabled

```
from itam.domain.asset import Asset

class AssetManager:
    def __init__(self):
        self._assets = []
        self.observers = []

    def add_observer(self, observer):
        self.observers.append(observer)

    def remove_observer(self, observer):
        self.observers.remove(observer)

    def notify_observers(self, event, asset):
        for observer in self.observers:
            observer.update(event, asset)

    def read(self, asset_id: int) -> Asset:
        for asset in self._assets:
            if asset.get_id() == asset_id:
                return asset
        raise ValueError("Asset not found")

    def create(self, asset: Asset) -> None:
        self.notify_observers("CREATE", asset)
        self._assets.append(asset)

    def update(self, asset: Asset) -> None:
        for i in range(len(self._assets)):
            if self._assets[i].get_id() == asset.get_id():
                self._assets[i] = asset
                self.notify_observers("UPDATE", asset)
                return
        raise ValueError("Asset not found")

    def delete(self, asset_id: int) -> None:
        for i in range(len(self._assets)):
            if self._assets[i].get_id() == asset_id:
                del self._assets[i]
                self.notify_observers("DELETE", asset_id)
                return
        raise ValueError("Asset not found")

    def get_assets(self):
        return self._assets
```

We will see two more patterns in the next section of this chapter: the Singleton Pattern and the Repository Pattern. However, before we see them, we will need to set up a Controller to interact with our system. That takes us to the usage of Ports and Adapters.

4.3 Plugging (in) Ports and Adapters

Our Information Technology Asset Management system is beginning to come together; the core business model has been built out. We have applied patterns to make the code more concise, readable, and maintainable. There is, however, one glaring deficiency: How do we interact with it? That is the subject of this section.

As you might recall hexagonal architecture is an approach to designing software that emphasizes the separation of the core business logic from the external services.

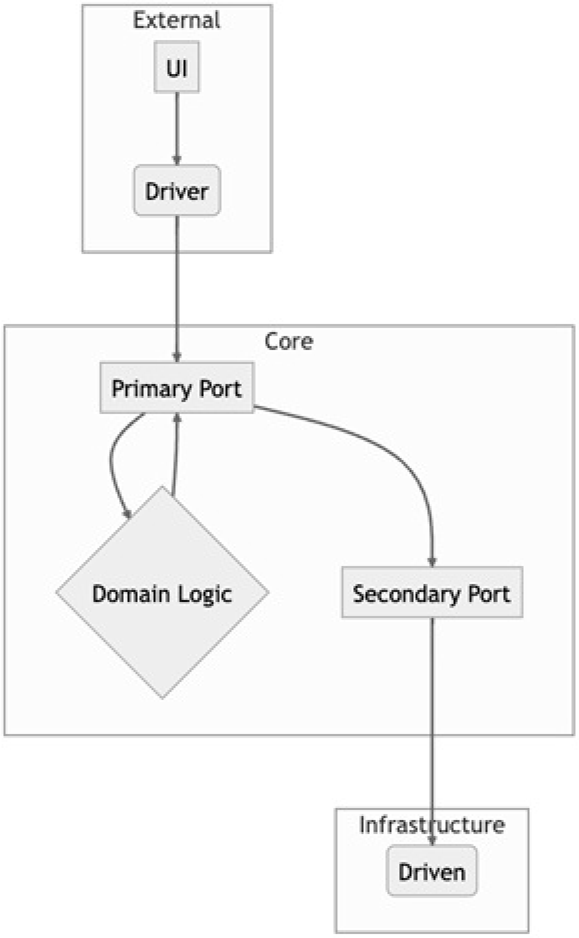
The business logic can be thought of as the "brain" of the application. It contains all the important rules and constructs that the application will need to guarantee the correctness of the program. In this analogy, then, the external services would be your "hands" or "eyes". They allow interactions with the outside world: user interfaces, database, etc.

Hexagonal architecture separates the main program logic from the outside parts like buttons, screens, and databases. It makes it easy to change those outside parts without changing the main program. It does this by using "ports" that define how the outside parts can interact with the main program, and "adapters" that make those interactions happen in a concrete way.

This approach makes it easier to change and evolve the application over time.

If a change needs to be made to one of the external systems, the application’s core should be unaffected. You would only need to update the adapter. The following diagram illustrates this:

Figure 4.7 A visualization of hexagonal architecture in action. Notice that the Core is segregated from the rest of the system and is only interacted with via Ports.



We will begin by building out a driver of the system. A driver is a system external to the context boundary of the application that sends requests to the system and, optionally, receives responses from the application; a conventional example being a REST call from a web browser to a REST controller.

First, we will add a REST controller to our Information Technology Asset Management system. The REST controller will expose the functionality provided by the `AssetManager` class. We will create a directory called infrastructure/API, in which we will create a file called `asset_controller.py`. At the beginning of this file add the following prompt.

Listing 4.29 The prompt to create a REST controller to expose the functionality of `AssetManager`

```
# Import statements for Asset, AssetService, AssetFactory, AssetI
# Use AssetFactory to create an Asset from an AssetIn
# Construct an instance of the APIRouter class from FastAPI
# Define HTTP endpoints to handle CRUD operations for the AssetSe
# Implementation of error handling, logging, and other necessary
# All methods should be asynchronous
```

Given that we did not specify the path to use, it is possible that Copilot will produce inconsistent or invalid paths with the routes. This was what was output in VS Code, using the prompt above. More of a preference than a standard, but the URLs should be plural.

Listing 4.30 The generated code’s route is “asset” rather than “assets”

```
@router.post('/asset', response_model=AssetOut)
async def create_asset(asset_in: AssetIn):
    asset = AssetFactory.create_asset(asset_in)
    asset_service.create(asset)
    return from_asset(asset)
```

You might elect to manually update the code to reflect your path preference. Above all, though, ensure that it is consistent across the various methods. The first listing will show us how to initialize all of the services that we will need to run the application.

Listing 4.31 Updated `AssetController` code to reflect consistent routes

```
from fastapi import APIRouter, HTTPException
from itam.domain.asset import Asset
from itam.service.asset_manager import AssetManager
from itam.domain.factory.asset_factory import AssetFactory
from itam.infrastructure.api.asset_model import AssetIn, AssetOut
import logging

router = APIRouter()
```

```
asset_manager = AssetManager()
asset_factory = AssetFactory()
```

Next, we will see how the routes have been defined and their methods get used. The first two routes define how we will access our `Asset` objects. The first GET request will get all `Assets` that we currently have in the system. In the real world, we would include convenience features such as pagination and sorting. However, given the limited number of entries in the system currently, we shall forgo this. The next GET method will get a specific `Asset` by its identifier.

Listing 4.32 The `AssetController` methods to access `Assets`

```
@router.get('/assets', response_model=list[AssetOut])
async def get_assets():
    assets = asset_manager.get_assets()
    return [from_asset(asset) for asset in assets]

@router.get('/assets/{asset_id}', response_model=AssetOut)
async def read_asset(asset_id: int):
    try:
        asset = asset_manager.read(asset_id)
        return from_asset(asset)
    except ValueError as e:
        logging.error(e)
        raise HTTPException(status_code=404, detail="Asset not fo
```

The final set of routes define how we will create, update, and remove `Assets` from our system. Note, that we are not doing “soft” deletes, which would just set a flag and this Asset would not be returned upon subsequent queries.

Listing 4.33 The `AssetController` methods to modify and delete `Assets`

```
@router.post('/assets', response_model=AssetOut)
async def create_asset(asset_in: AssetIn):
    asset = asset_factory.new(asset_in.asset_type, asset_in.name,
    asset_manager.create(asset)
    return from_asset(asset)

@router.put('/assets/{asset_id}', response_model=AssetOut)
async def update_asset(asset_id: int, asset_in: AssetIn):
    try:
        asset = asset_factory.new(asset_in.asset_type, asset_in.n
        asset.set_id(asset_id)
        asset_manager.update(asset)
        return from_asset(asset)
    except ValueError as e:
        logging.error(e)
        raise HTTPException(status_code=404, detail="Asset not fo

@router.delete('/assets/{asset_id}')
async def delete_asset(asset_id: int):
    try:
        asset_manager.delete(asset_id)
    except ValueError as e:
        logging.error(e)
        raise HTTPException(status_code=404, detail="Asset not fo
```

You should note within that class that there are simplified representations of the `Asset` class called, not coincidently, `AssetIn` and `AssetOut`. You could define those classes within the `AssetController` or you could add a separate class and define them there. In the reference implementation for this book, they are defined in a file called `asset_model.py`.

Listing 4.34 The prompt to define the `AssetIn` and `AssetOut` classes

```
# Define a pydantic model for AssetIn
# It should have the following attributes:
#    name: str
#    asset_type: str
#    department: str
#    depreciation_strategy: str
#    useful_life: int
#    unit_cost: float
#    depreciation_rate: float
#    salvage_value: float

# Define a pydantic model for AssetOut
# It should have the following attributes:
#    id: int
#    name: str
#    asset_type: str
#    department: str
#    depreciation_strategy: str
#    useful_life: int
#    unit_cost: float
#    depreciation_rate: float
#    salvage_value: float
# It should have a method that transforms an Asset into an AssetO
```

With the prompt defined at the beginning of the file, we should get code generated by Copilot that is similar to the code in listing 4.34.

Listing 4.35 The classes `AssetIn` and `AssetOut` in `asset_model.py`

```
from pydantic import BaseModel
from itam.domain.asset import Asset

class AssetIn(BaseModel):
    name: str
    asset_type: str
    department: str
    depreciation_strategy: str
    useful_life: int
    unit_cost: float
    depreciation_rate: float
    salvage_value: float
    purchase_date: str

class AssetOut(BaseModel):
    id: int
    name: str
    asset_type: str
    depreciation_strategy: str
    useful_life: int
    unit_cost: float
    depreciation_rate: float
    salvage_value: float
    purchase_date: str

@staticmethod
def from_asset(asset: Asset) -> AssetOut:
    return AssetOut(
        id=asset.id,
        name=asset.name,
        asset_type=asset.category,
        depreciation_strategy=str(asset.funding_details.depreciat
        useful_life=asset.useful_life,
        unit_cost=asset.cost,
        depreciation_rate=asset.funding_details.depreciation_rate
        salvage_value=asset.salvage_value,
        purchase_date=asset.purchase_date.strftime("%Y-%m-%d"),
    )
```

We could, optionally, add another layer of abstraction between the `AssetController` and the AssetManager. One might consider during so might be hexagonal architecture in extremis, but there are few challenges in Computer Science that cannot be resolved by adding one more layer of abstraction. We will add a Port that will be used to interface with the `AssetController`. Let’s create a prompt for the class `AssetRestPort` and have Copilot generate the code.

Listing 4.36 A prompt to create the `AssetRestPort`

```
# Define an interface called AssetRestPort
# It should expose all of the methods in AssetManager as abtract
```

This prompt instructs Copilot to create an interface that we will implement shortly.

Listing 4.37 Complete source code for `AssetRestPort`

```
from itam.domain.asset import Asset
from abc import abstractmethod

class AssetRestPort:
    @abstractmethod
    def read(self, asset_id: int) -> Asset:
        pass

    @abstractmethod
    def create(self, asset: Asset) -> None:
        pass

    @abstractmethod
    def update(self, asset: Asset) -> None:
        pass

    @abstractmethod
    def delete(self, asset_id: int) -> None:
        pass

    @abstractmethod
    def get_assets(self):
        pass
```

Next, we will implement this Port with an Adapter that will be used to interaact with the Controller to further abstract the components. We would be able to swap out this Adapter with another, should we want to turn this into a command line application, for example. The prompt for the Adapter is as follows.

Listing 4.38 A prompt to create the `AssetRestAdapter`

```
# Define an interface called AssetRestAdapter
# It should take an AssetManager as a parameter in its constructo
# It should expose all of the methods in AssetManager
# It should inherit from AssetRestPort
```

There are two important elements to this prompt. The first is that it implements the Port interface that we had previously defined. The second is that it takes the `AssetManager` and wraps its functionality.

Listing 4.39 The source code for AssetRestAdapter

```
from itam.domain.asset import Asset
from itam.infrastructure.ports.asset_rest_port import AssetRestPo
from itam.service.asset_manager import AssetManager

class AssetRestAdapter(AssetRestPort):
    def __init__(self, asset_manager: AssetManager):
        self._asset_manager = asset_manager

    def read(self, asset_id: int) -> Asset:
        return self._asset_manager.read(asset_id)

    def create(self, asset: Asset) -> None:
        self._asset_manager.create(asset)

    def update(self, asset: Asset) -> None:
        self._asset_manager.update(asset)

    def delete(self, asset_id: int) -> None:
        self._asset_manager.delete(asset_id)

    def get_assets(self):
        return self._asset_manager.get_assets()
```

All that is left to do is to update the `AssetController` to remove the direct invocation of the AssetManager methods by the `AssetController`, and rather have the `AssetController` invoke the methods of the Adapter, which in turn invokes the methods of the `AssetManager`. The key takeaway from the Ports and Adapters pattern is that it abstracts the interaction between the driving parts of the system, in this case the REST API, and the driven application: the business model and core of our system: the `AssetManager`. To make this more explicit and as a little preview, we will shortly modify this class yet again to add the Port to the constructor.

Listing 4.40 Updated code for `AssetController` which uses the `AssetRestAdapter`

```
router = APIRouter()

asset_rest_adapter = AssetRestAdapter(AssetManager())
asset_factory = AssetFactory()

@router.post('/assets', response_model=AssetOut)
async def create_asset(asset_in: AssetIn):
    asset = asset_factory.new(asset_in.asset_type, asset_in.name,
    asset_rest_adapter.create(asset)
    return from_asset(asset)
```

As previously mentioned, we will modify the `AssetController` to remove all direct references to the AssetManager. Our current `AssetController`, while it does not directly invoke any AssetManager methods, does hold an indirect reference to the `AssetManger`, as the AssetManager is constructed in the `AssetController`. Doing so will further insulate the AssetManager from the changes of the drivers.

Abstractions in software are not merely an academic matter. These abstractions allow one flexibility in how one can satisfy the requirements and make changes easier to make. They can hide complexity, encapsulating behavior behind seemingly simple interfaces. This will bring us to our next pattern: the *Repository Pattern*.

The Repository Pattern

The Repository pattern is a design pattern that helps manage data by creating a separate place to handle tasks like getting, saving, updating, and deleting data. It connects the data source (like a database or file) to the rest of the application, keeping data access code apart from other code.

In our ITAM project, the Repository Pattern will allow us to present a greatly simplified interface, masking the complexity of, in this case, our data access. From the perspective of the Repository or the AssetManager for that matter, our data access layer is just a series of methods. Behind the scenes it might be a REST API, a Redis cache, MongoDb, a file, or a database of any flavor. Let’s create a prompt that will set up our generic data access layer, using the Repository Pattern.

Listing 4.41 The prompt to create our generic data access layer

```
# Create an abstract base class called BaseRepository that repres
# The class should define common methods for CRUD operations such
# Make sure to use type hints for the input parameters and return
```

With this prompt, Copilot should have everything it is needs to establish the contract by which all implementations will honor and behave. Let’s examine the code.

Listing 4.42 The code for the `BaseRepository` which will function as our generic data access layer

```
from abc import ABC, abstractmethod
from typing import List, TypeVar, Generic

T = TypeVar('T')

class BaseRepository(ABC, Generic[T]):
    @abstractmethod
    def create(self, entity: T) -> None:
        pass

    @abstractmethod
    def read(self, entity_id: int) -> T:
        pass

    @abstractmethod
    def update(self, entity: T) -> None:
        pass

    @abstractmethod
    def delete(self, entity_id: int) -> None:
        pass

    @abstractmethod
    def get_entities(self) -> List[T]:
        pass
```

Copilot introduced a Generic type, allowing for some typing hints when it comes time to implement. If we express that the type should be an `Asset`, then the method signature would demonstrate this. Our first attempt at an implementation will be an in-memory data store. This will allow us to perform an end-to-end test of all of our system. It also nicely demonstrates why abstractions are useful. Before long, we will add another implementation that uses Postgres. But first, the prompt:

Listing 4.43 The prompt to an in-memory data store for testing purposes

```
# Define a class called InMemoryAssetRepository that inherits fro
# The InMemoryAssetRepository should be specific to managing Asse
# Implement the necessary methods for CRUD operations such as cre
# Make sure to use type hints for the input parameters and return
```

Our in-memory implemenation will use a simple list in which to store the `Assets`. We will be able to perform all CRUD operations. Additionally, Copilot will add in the typing hints for us: letting others know that it should accept and return Assets. That is what is meant by `BaseRepository[Asset]`: this is an implementation of a `BaseRepository` whose Generic Type is Asset.

Listing 4.44 The code for the `InMemoryAssetRepository`

```
from itam.infrastructure.repository.base_repository import BaseRe
from itam.domain.asset import Asset

class InMemoryAssetRepository(BaseRepository[Asset]):
    def __init__(self):
        self._assets = []

    def create(self, asset: Asset) -> None:
        self._assets.append(asset)

    def read(self, asset_id: int) -> Asset:
        return next((asset for asset in self._assets if asset.id

    def update(self, asset: Asset) -> None:
        for i in range(len(self._assets)):
            if self._assets[i].id == asset.id:
                self._assets[i] = asset
                break

    def delete(self, asset_id: int) -> None:
        self._assets = [asset for asset in self._assets if asset.

    def get_entities(self) -> list[Asset]:
        return self._assets
```

Finally, we will update the `AssetManager` to delegate the CRUD operations of the `Assets` to a `BaseRepository` instance (`_repository`). The full source code is below, including the prompt, located at the beginning of the file.

Listing 4.45 The updated prompt and code for the `AssetManager` to use the `InMemoryAssetRepository`

```
# Define a class called AssetManager
# with CRUD operations for Asset
# and a method to notify observers whenever an asset is created,
# The AssetManager should use an instance of the InMemoryAssetRep
# Implement methods for creating, reading, updating, and deleting
# Please include type hints for the input parameters and return v
# The methods should be named create, read, update, get_assets, a

from itam.domain.asset import Asset
from itam.infrastructure.repository.in_memory_asset_repository im

class AssetManager:
    def __init__(self):
        self._repository = InMemoryAssetRepository()

    def create(self, asset: Asset) -> Asset:
        self._repository.create(asset)
        return asset

    def read(self, asset_id: int) -> Asset:
        return self._repository.read(asset_id)

    def update(self, asset: Asset) -> Asset:
        self._repository.update(asset)
        return asset

    def get_assets(self) -> list[Asset]:
        return self._repository.get_entities()

    def delete(self, asset_id: int) -> None:
        self._repository.delete(asset_id)
```

At this point, we have a core business domain that is untouched directly by our system. We have ports by which requests can come in. We also have ports by which we can store the data (at least while the system is running). We should be able to test the system end-to-end, by running it and POST-ing request to the creation end point, and the reading from the GET end points. Once we have confirmed that the system works end-to-end, we can now tackle the issue that our data is only persisted in memory; we can now hook up an actual database. In order to do this, we introduce the final pattern of this chapter: the *Singleton*. The Singleton, conceptually, is very simple to grok; there should only ever be one instance of it running. The Singleton is good for many use cases: logging, caching, configuration settings, or a database connection manager.

The Singleton Pattern

The Singleton pattern is a design pattern that ensures a class has only one instance and provides a global access point to that instance. It is used when you want to have a single object shared across different parts of your program, instead of creating multiple instances of the same class.

We will ask Copilot to create a Singleton class for us to manage our database's connections. Since we never want to hard code usernames or passwords (or any connection details for that matter) in our source code, not only because it is inherently less secure, but the details will likely differ depending on the environment (DEV versus QA versus PROD). We will therefore ask Copilot to accept these values as environmental variables and configure the connections using these values.

Listing 4.46 A prompt to create the Singleton `DatabaseConnection` class

```
# Create a Python class called DatabaseConnection using the Singl
# The class should read the database username, password, and conn
# The environment variables should be named: DB_USERNAME, DB_PASS
```

We have asked Copilot to employ SQLAlchemy, an Object Relational Mapping (ORM) tool, to perform the actions against the database. Copilot will adroitly assemble our source code. The `@staticmethod` keyword will create a method that belongs to class rather than the instance of the class. This keyword is used to get the instance of the `DatabaseConnection` class. Since a static method cannot modify the instance data, it can be used to ensure that only one instance of this class is ever running, and thus is a Singleton. The constructor initializes the connection to the database using string interpolation using environment variables.

Listing 4.47 A prompt to create the Singleton `DatabaseConnection` class

```
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker
import os

class DatabaseConnection:
    __instance = None
    @staticmethod
    def get_instance():
        if DatabaseConnection.__instance == None:
            DatabaseConnection()
        return DatabaseConnection.__instance

    def __init__(self):
        if DatabaseConnection.__instance != None:
            raise Exception("This class is a singleton!")
        else:
            DatabaseConnection.__instance = self
            self.engine = create_engine(f"postgresql://{os.enviro
            self.Session = sessionmaker(bind=self.engine)

    def get_session(self):
        return self.Session()
```

Now that now get a connection to our database, we will need to create a repository that will use this connection to store and retrieve our Asset objects. We will create a class called `SQLAlchemyAssetRepository`. This name very explicitly states what it is and how it does it.

Listing 4.48 A prompt and code to the `SQLAlchemyAssetRepository` class

```
# Define a class called SQLAlchemyAssetRepository that implements
# The class should handle the CRUD operations (create, read, upda
from itam.domain.asset import Asset
from itam.infrastructure.repository.base_repository import BaseRe
from sqlalchemy.orm import Session

class SQLAlchemyAssetRepository(BaseRepository[Asset]):
    def __init__(self, session: Session):
        self._session = session

    def create(self, asset: Asset) -> Asset:
        self._session.add(asset)
        self._session.commit()
        return asset

    def read(self, asset_id: int) -> Asset:
        return self._session.query(Asset).filter(Asset.id == asse

    def update(self, asset: Asset) -> Asset:
        self._session.query(Asset).filter(Asset.id == asset.id).u
        self._session.commit()
        return asset

    def get_assets(self) -> list[Asset]:
        return self._session.query(Asset).all()

    def delete(self, asset_id: int) -> None:
        self._session.query(Asset).filter(Asset.id == asset_id).d
        self._session.commit()
```

Next, we will refactor the `AssetController` to allow us to pass in the `AssetRestPort`. By doing this, we can swap out the driven aspect of our application. This refactor will further abstract the driver from the business core.

Listing 4.49 The updated `AssetController` class which exposes the routes rather than using the method decorators

```
from itam.infrastructure.ports.asset_rest_port import AssetRestPo
import logging

class AssetController:
    def __init__(self, asset_rest_port:  AssetRestPort):
        self._asset_factory = AssetFactory()
        self._asset_rest_port = asset_rest_port
        ...

    def get_router(self):
        return self._router

    async def get_assets(self):
        return  [ from_asset(a) for a in self._asset_rest_port.ge

    async def get_asset(self, asset_id: int):
        asset = self._asset_rest_port.read(asset_id)
        if asset is None:
            raise HTTPException(status_code=404, detail="Asset no
        return from_asset(asset)

    async def create_asset(self, asset_in: AssetIn):
        asset = self._asset_factory.new(asset_in.asset_type, asse
        self._asset_rest_port.create(asset)
        return from_asset(asset)

    async def update_asset(self, asset_id: int, asset_in: AssetIn
        asset = self._asset_factory.new(asset_in.asset_type, asse

        asset.id = asset_id
        asset = self._asset_rest_port.update(asset)
        if asset is None:
            raise HTTPException(status_code=404, detail="Asset no
        return from_asset(asset)

    async def delete_asset(self, asset_id: int):
        asset = self._asset_rest_port.read(asset_id)
        if asset is None:
            raise HTTPException(status_code=404, detail="Asset no
        self._asset_rest_port.delete(asset_id)
        return from_asset(asset)
```

We can now consolidate the initialization logic of our application into the `main.py` file. This is the big payoff. Our system will have layering, facilitating the swapping out of the components are needed or as requirements change.

Listing 4.50 The finalized version of the `main.py` class, in which we wire our application together

```
from fastapi import FastAPI
from itam.infrastructure.api.asset_controller import AssetControl
#from itam.infrastructure.repository.in_memory_asset_repository i
from itam.infrastructure.repository.sqlalchemy_asset_repository i
from itam.infrastructure.database.database_connection import Data
from itam.service.asset_manager import AssetManager
from itam.infrastructure.adapters.asset_rest_adapter import Asset
```

```
import uvicorn

app = FastAPI()
session = DatabaseConnection().get_session()
#repository = InMemoryAssetRepository()
repository = SQLAlchemyAssetRepository(session)
asset_manager = AssetManager(repository)
asset_rest_adapter = AssetRestAdapter(asset_manager)
asset_controller = AssetController(asset_rest_adapter)
app.include_router(asset_controller.get_router())

if __name__ == '__main__':
    uvicorn.run(app, host='0.0.0.0', port=8000)
```

Congratulations, we now have a running system, which is persisting data to our database.

4.4 Summary

- The Decorator Design Pattern is a structural design pattern that allows you to add new or modify existing behavior to an object dynamically without changing the existing class. This is done by wrapping the current object in the decorating object.
- Visitor pattern to add new behavior to or change the existing behavior of a given class.
- The Factory pattern is another creation pattern that allows you to abstract away some of the details of the object that you are attempting to create.
- The Builder Pattern is a creational design pattern provides a fluent API for the creation of objects by providing step by step instructions on how to create said object.
- The Adapter Pattern is a structural design pattern that allows one to bridge the gap between the target interface and a class with an incompatible interface.
- The Observer Pattern is a behavioral pattern in which the Subject class reports certain state changes to Observer classes via notifications.
- Hexagonal architecture separates the main program logic from the outer parts like buttons, screens, and databases. It makes changing outer parts easy without changing the main program.
- The Repository pattern is a design pattern that helps manage data by

creating a separate place to handle tasks like getting, saving, updating, and deleting data. It connects the data source (like a database or file) to the rest of the application, keeping data access code apart from other code.

- The Singleton pattern is a design pattern that ensures a class has only one instance and provides a global access point to that instance. It is used when you want to have a single object shared across different parts of your program instead of creating multiple instances of the same class.