ABSTRACTION LAYERS IN PROGRAMMING: AN OVERVIEW



This article explains abstraction layers, which you can use in many programming domains. Let's get started.

What is an abstraction layer?

The abstraction layer creates a separation between two things. For programming, this is often splitting tasks into separate entities.

For example, an entity A might be assigned the task of fetching an image from a database and then processing the image when retrieved

Creating an abstraction layer will split this entity A into entities A and B, where:

- Entity A fetches the resource.
- Entity B—now the abstraction—depends on the image to return from the first operation to then perform its second operation.

The abstraction layer allows one party, or operation, to be entirely responsible for one task while a dependent waits for that party to return data for them to process.



(Source)

Abstraction in code

APIs are abstraction layers. While APIs serve many purposes, the way they are used is this: a developer is trying to create an application involving tweets, so they tap into Twitter's new APIv2 to get tweet data.

The developer is not responsible for creating a service that allows users to speak with one another via 140-character messages. The Twitter service is operated by Twitter, and all responsibility for collecting their data rests on their shoulders.

The developer just has to to get the data and use it for their purposes. Because the developer is not responsible for getting the data and providing the Twitter service at their level of operability, the Twitter API is abstracted away from the developer's operation.

In our case, abstraction can also be called indirection. In a famous quote by David Wheeler,

> *"All problems in computer science can be solved by another level of indirection."*

Creating an abstraction is as simple as changing one function into two. This Python function opens a text document and counts the number of lines, the number of words, and the number of characters it has.

```
def processDoc(filename):
with open(filename,'r') as f
```

```
number_of_lines = 0
number_of_words = 0
number_of_characters = 0
for line in f:
line = line.strip("\n")
words = line.split()
number_of_lines += 1
number_of_words += len(words)
number_of_characters += len(line)
return number_of_lines, number_of_words, number_of_characters
```

To create an abstraction layer from this function, the "counting" portion of the function can get abstracted away from the "opening" portion of the function. One function is not responsible for both; two functions take responsibility for the process:

- The **openDoc** function will open the document and return the document as a string that the computer can use.
- The **processDoc** function takes as input, the string output from the openDoc function, and runs some counting scripts and returns their values.

To the original function, the step that processes the texts is the abstraction layer—it is one step removed from the original task. Using the appropriate nomenclature, it has been *abstracted* away from the original function.

```
def openDoc(filename):
with open(filename,'r') as f
text = f.readlines
return text

def processDoc(text):
number_of_lines = 0
number_of_words = 0
number_of_characters = 0
for line in text:
line = line.strip("\n")
words = line.split()
number_of_lines += 1
number_of_words += len(words)
number_of_characters += len(line)
return number_of_lines, number_of_words, number_of_characters
```

Abstraction layers in different domains

You'll use abstraction layers differently depending on your programming domain.

For front end programming, creating more modular code means creating more abstraction layers. Using modular code in front end programs enables the code's:

- Reusability

- Long-term sustainability

Containers and, more so, the use of containers for specific purposes in the code uses abstraction layers. For back-end developers, containers are their way to make code more modular.

Functions as a service (FaaS), which are usually functions wrapped into a container, are functions that operate an abstraction level away from the main code. The code example above are two functions that could be written into two different containers:

- One function is a cloud function that opens files and returns the text from it.
- The other can be a cloud function that takes a string as input, and counts its lines, words, and characters.

In Kubeflow, Google's open source machine learning software, every step in the pipeline is a container. Instead of running an ML job all from one Python script, inside one Jupyter notebook, or on one container, each step in the file gets converted to a function of inputs and outputs. The entire notebook is pushed to abstraction and run through containers. Using a built-in Kubeflow method, they provide the tools to convert a function into a container.

In its design, the Kubernetes infrastructure, an ability to maneuver and run containers on an immutable infrastructure, advocates for code functions and processes to exist in abstracted layers.

When to use abstraction layers

Abstraction layers are not necessary for beginning programmers. Abstract layers come in play as more of an artform—they can be used when thinking about the design of the code, and to determine which processes happen when and where, and which processes depend on each other.

Additional resources

For related reading, explore these resources:

- BMC DevOps Blog
- Resilience Engineering: An Introduction
- BMC Machine Learning & Big Data Blog
- Containerized Machine Learning: An Intro to ML in Containers