Layered Architecture

February 28, 2022

Brae Webb

Presented for the Software Architecture course
at the University of Queensland

THE UNIVERSITY OF QUEENSLAND
AUSTRALIA

Last Updated on 2022/01/21

Layered Architecture

Software Architecture

February 28, 2022

Brae Webb

1 Introduction

In the beginning there was the *big ball of mud*. The ball of mud is an architectural style identified by it's lack of architectural style [1]. In a ball of mud architecture, all components of the system are allowed to communicate. If your GUI code wants to ask the database a question, it will write an SQL query and ask it. Likewise, if the code which primarily talks to the database decides your GUI needs to be updated a particular way, it will make it so.

The ball of mud style is a challenging system to work under. Modifications can come from any direction at any time. Akin to a program which primarily uses global variables, it is hard, if not impossible, to understand everything that is happening or could happen.

TODO: Monolith is technically before layered architecture and monolith $\neq$ big ball of mud

Aside

Code examples in these notes are works of fiction. Any resemblance to a working code is pure coincidence. Having said that, python-esque syntax is often used for it's brevity. We expect that you can take away the important points from the code examples without getting distracted in the details.

```
import gui
import database

button = gui.make_button("Click me to add to counter")
button.onclick(e =>
    database.query("INSERT INTO clicks (time) VALUES {{e.time}}"))
```

Figure 1: A small example of a *big ball of mud* architecture. This type of software is fine, even preferred, for smaller projects. However, it does not work well at scale.

The first architectural style we will investigate is a layered architecture. Layered architecture (also called multi-tier or tiered architecture) partitions software into specialised components and restricts how those components can communicate with each other. A layered architecture creates superficial boundaries between the software components, or layers. Often component boundaries aren't enforced by the technology but by architectural policy.

The creation of these boundaries provides the beginnings of some control over what your software is allowed to do. Communication between the component boundaries is done via well-specified *contracts*. The use of contracts results in each layer knowing precisely how it can be interacted with. Furthermore, when a layer needs to be replaced or rewritten, it can be safely substituted with a layer fulfilling the contract.

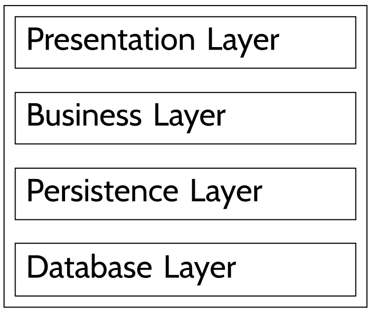

Figure 2: The traditional specialised components of a layered architecture.

2 Standard Form

The traditional components of a layered architecture are seen in Figure 2. This style of layered architecture is the four-tier architecture. Here, our system is composed of a presentation layer, business layer, persistence layer, and database layer.

The presentation layer takes data and formats it in a way that is sensible for humans. For command line applications, the presentation layer would accept user input and print formatted messages for the user. For traditional GUI applications, the presentation layer would use a GUI library to communicate with the user.

The business layer is the logic central to the application. The interface to the business layer is events or queries triggered by the presentation layer. It's the responsibility of the business layer to determine the data updates or queries required to fulfil the event or query.

The persistence layer is essentially a wrapper over the database, allowing more abstract data updates or queries to be made by the business layer. One advantage of the persistence layer is it enables the database to be swapped out easily.

Finally, the database layer is normally a commercial database application like MySQL, Postgres, etc. which is populated with data specific to the software. Figure 3 is a over-engineered example of a layered architecture.

```
» cat presentation.code

import gui
import business

button = gui.make_button("Click me to add to counter")
button.onclick(business.click)
```

```
» cat business.code

import persistence

def click():
    persistence.click_counts.add(1)
```

Figure 3: An unnecessarily complicated example of software components separated into the standard layered form.

```
» cat persistence.code

import db

class ClickCounter:
    clicks = 0

    def constructor():
        clicks = db.query("SELECT COUNT(*) FROM clicks")

    def get_click():
        return clicks

    def add(amount):
        db.query("INSERT INTO clicks (time) VALUES {{time.now}}")

click_counts = ClickCounter()
```

Figure 3: An unnecessarily complicated example of software components separated into the standard layered form.

One of the key benefits afforded by a well designed layered architecture is each layer should be interchangeable. A typical example is an application which starts as a command line application but can later be adapted to a GUI application by just replacing the presentation layer.

3 Deployment Variations

While the layered architecture is popular with software deployed on one machine (a non-distributed system), layered architectures are also often deployed to separate machines.

Each layer can be deployed to separate binaries on separate machines. The most common variant of distributed deployment is separating the database layer. Since databases have well defined contracts and are language independent, the database layer is a natural first choice for physical separation.

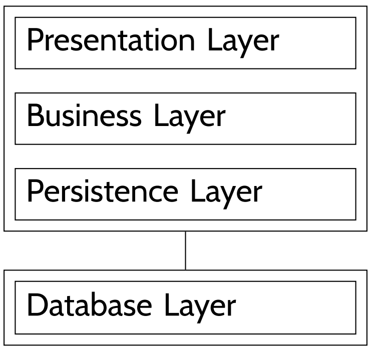

Figure 4: Traditional layered architecture with a separately deployed database.

Of course, in a well designed system, any layer of the system could be physically separated with minimal difficulty. The presentation is another common target. Physically separating the presentation layer gives users the ability to only install the presentation layer and allow communication to other software components to occur via network communication.

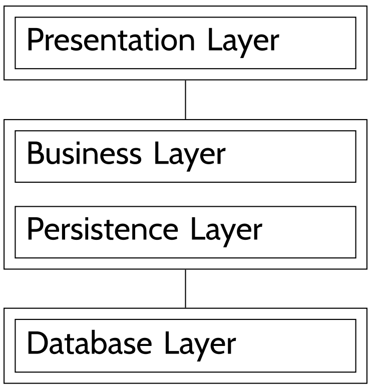

Figure 5: Traditional layered architecture with a separately deployed database and presentation layer.

This deployment form is very typical of web applications. The presentation layer is deployed as a HTML/JavaScript application which makes network requests to the remote business layer. The business layer then validates requests and makes any appropriate data updates.

A much more uncommon deployment variation (Figure 6) separates the presentation layer and business layer from the persistence layer and database layer. An updated version of our running example is given in Figure 7, the presentation layer remains the same but the communication between the business and persistence is now via REST.[1]

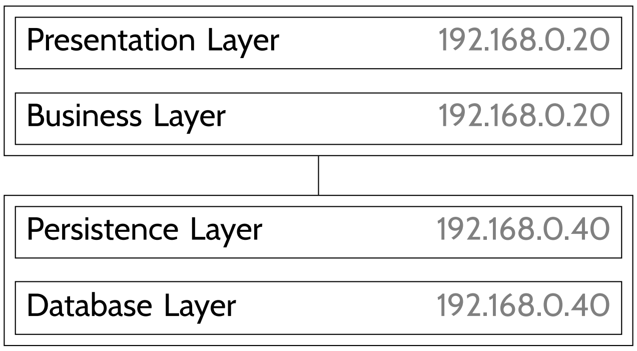

Figure 6: A contrived example of a deployment variation.

```
» cat business.code
import http

def click():
    http.post(
        address="192.168.0.40",
        endpoint="/click/add",
        payload=1
    )
```

Figure 7: Code adapted for the contrived example of a deployment variation.

[1]https://restfulapi.net/

```
» cat persistence.code

import db
import http

class ClickCounter:
    ... # as above

click_counts = ClickCounter()

http.on(
    method="post",
    endpoint="/click/add",
    action=(payload => click_counts.add(payload))
)
```

Figure 7: Code adapted for the contrived example of a deployment variation.

4 Closed/Open Layers

Separating software into layers helps to increase the modularity and isolation of distinct components. But, of course, components must communicate via their specified contracts, otherwise they wouldn't be particularly useful. Flows of communication between components are categorized as either *open* or *closed*. A layer, by default, is considered *closed*. Closed layers prevent the direct communication between their adjacent layers. Figure 8 shows the communication channels (as arrows) in a completely closed architecture.

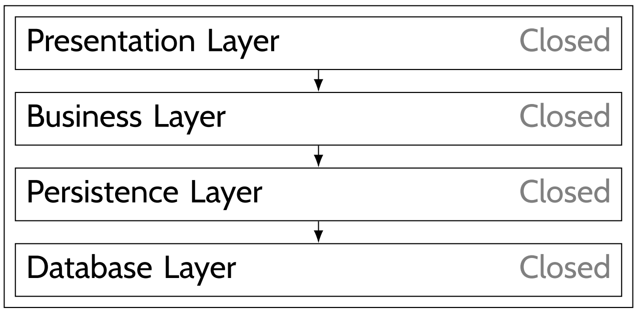

Figure 8: All layers closed requiring communication to pass through every layer.

An architecture where all layers are closed enables maximum isolation. A change to the communication contracts of any layer will require changes to at most one other layer. However, there are a number of situations where an *open* layer can be useful. Open layers do not require communication to pass through, communication can occur 'around' the layer.

TODO: Thinking about it, I'm not sure that open and closed layers is worth teaching. I think variations in the physical structures of layers which enables sidecars, etc from Bass would be better. It's more flexible and enables more obvious examples, i.e. a logging service sidecar

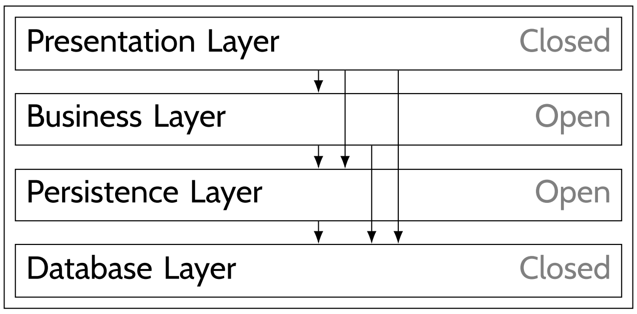

Figure 9: A wolf in layer's clothing [2]

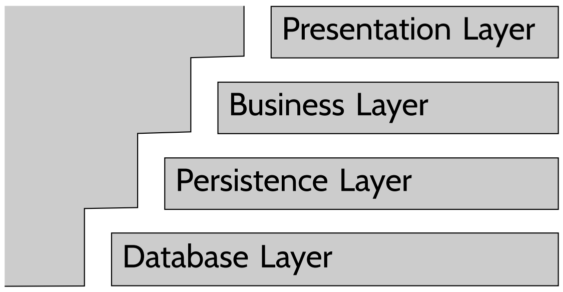

Figure 10: A more complicated layered architecture [2]

TODO: Figure is an example of a variation on open/closed layer model (fig 13.2 in bass - reconstruction in progress)

References

[1] B. Foote and J. Yoder, "Big ball of mud," *Pattern languages of program design*, vol. 4, pp. 654–692, 1997.

[2] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*. Addison-Wesley Professional, 3 ed., September 2012.