Layered Architecture

February 28, 2022

Brae Webb

Presented for the Software Architecture course
at the University of Queensland

THE UNIVERSITY
OF QUEENSLAND
AUSTRALIA

Layered Architecture

Software Architecture

February 28, 2022

Brae Webb

1 Introduction

In the beginning developers created the *big ball of mud*. It was without form and void, and darkness was over the face of the developers[1]. The big ball of mud is an architectural style identified by it's lack of architectural style [1]. In a big 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 do 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.

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 for experimentation but not for any code that has to be maintained. However, it does not work well at scale.

2 Monolith Architecture

And architects said, "let there be structure", and developers saw that structure was good. And architects called the structure *modularity*[2].

The monolithic software architecture is a single deployable application. There is a single code-base for the application and all developers work within that code-base. An example monolith application would

[1] Liberties taken from Genesis 1:1-2.
[2] Liberties taken from Genesis 1:3-5.

be one of the games developed by DECO2800[3] students at UQ[4]. (e.g. Retroactive[5]). A monolith should follow design conventions and be well structured and modular (i.e. it is not a big ball of mud).

Most developers are introduced to the monolith implicitly when they learn to program. They are told to write a program, and it is a single executable application. This approach is fine, even preferred, for small projects. It often becomes a problem for large, complex software systems.

2.1 Advantages

The advantages of a monolith are that it is easy to develop, deploy and test. A single code-base means that all developers know where to find all the source code for the project. They can use any IDE for development and simple development tools can work with the code-base. There is no extra overhead that developers need to learn to work on the system.

Being a single executable component, deployment is as simple as copying the executable on to a computer or server.

System and integration testing tends to be easier with a monolith, as end-to-end tests are executing on a single application. This often leads to easier debugging once errors are found in the software. All dependencies and logic are within the application.

There are also fewer issues to do with logging, exception handling, monitoring, and even scalability if it is running on a server.

2.2 Disadvantages

The drawbacks of a monolith are complexity, coupling and scalability. Being a single application, as it gets larger and more complex, there is more to understand. It becomes harder to know how to change existing functionality or add new functionality without creating unexpected side effects. A catch phrase in software design and architecture is to build complex systems, but not complicated systems. Monoliths usually become complicated as they grow to deliver complex behaviour.

Related to complexity is coupling, with all behaviour implemented in one system there tends to be greater dependency between different parts of the system. The more dependencies that exist, the more difficult it is to understand any one part of the system. This means it is more difficult to make changes to the system or to identify the cause of defects in the system.

A monolith running on a server can be scaled by running it on multiple servers. Because it is a monolith, without dependencies on other systems, it is easy to scale and replicate the system. The drawback is that you have to replicate the entire system on another server. You cannot scale components of the system independently of each other. If the persistence logic is creating a bottleneck, you have to copy the entire application on to another server to scale the application. You cannot user servers that are optimised to perform specialised tasks.

3 Layered Architecture

And architects said, "let there be an API between the components, and let it separate component from component[6]".

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

[3] https://my.uq.edu.au/programs-courses/course.html?course_code=DECO2800
[4] https://www.uq.edu.au/
[5] https://github.com/UQdeco2800/2021-studio-7
[6] Liberties taken from Genesis 1:6-8.

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.

3.1 Standard Form

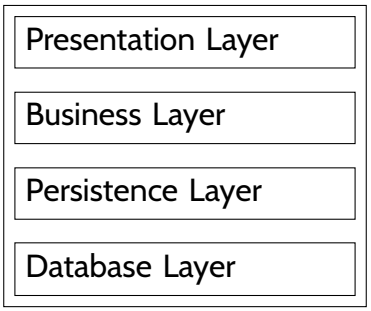

Figure 2: The traditional specialised components of a layered architecture.

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 an 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)
```

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

```
» cat business.code
import persistence

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

```
» 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.

3.2 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 as separate binaries on separate machines. A simple, common variant of distributed deployment is separating the database layer, as shown in figure 4. Since databases have well defined contracts and are language independent, the database layer is a natural first choice for physical separation.

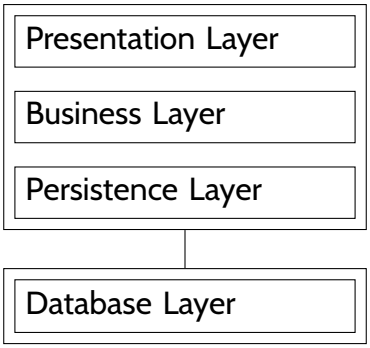

Figure 4: Traditional layered architecture with a separately deployed database.

In a well designed system, any layer of the system could be physically separated with minimal difficulty. The presentation layer is another common target, as shown in figure 5. 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.

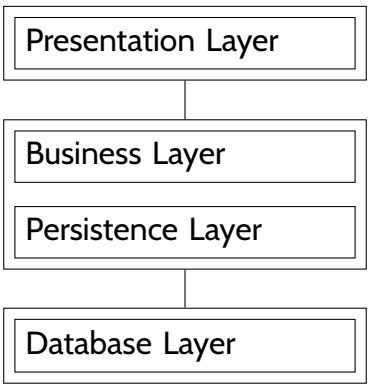

Figure 5: Traditional layered architecture with a separately deployed database and presentation layer.

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.

Some database driven appliation generators will separate embedded the application logic in the database code so that all logic runs on the database server. The presentation layer is then separated from the application logic, as shown in figure 6.

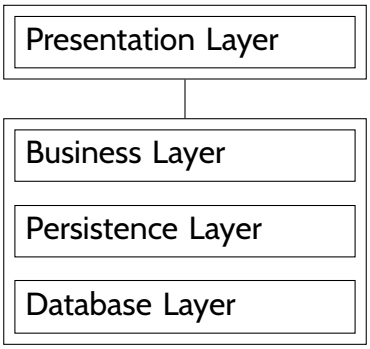

Figure 6: Traditional layered architecture with a separately deployed presentation layer.

An uncommon deployment variation (figure 7) separates the presentation and business layers from the persistence and database layers. An updated version of our running example is given in figure 8, the

presentation layer remains the same but the communication between the business and persistence layers is now via REST.[7]

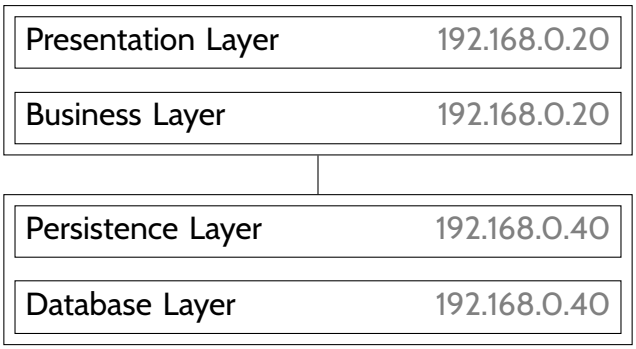

Figure 7: A contrived example of a deployment variation.

```
» cat business.code
```

```
import http

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

```
» 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 8: Code adapted for the contrived example of a deployment variation.

[7] https://restfulapi.net/

3.3 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 9 shows the communication channels (as arrows) in a completely closed architecture.

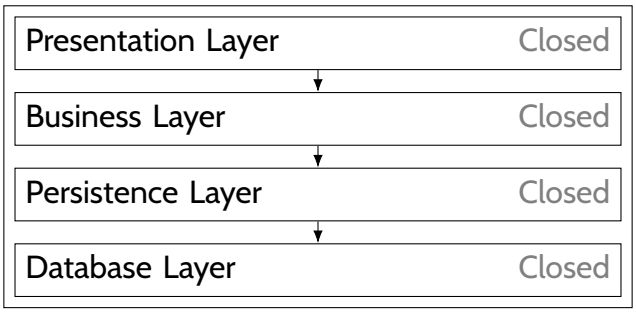

Figure 9: All layers closed requiring communication to pass through every layer.

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

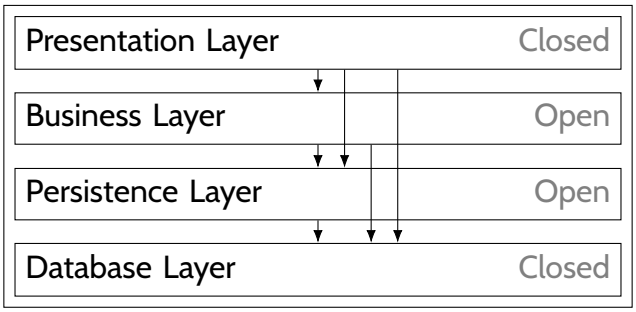

Figure 10: A wolf in layer's clothing [2]

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.

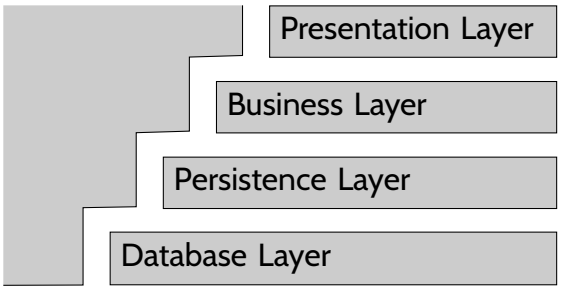

Figure 11: A more complicated layered architecture [2]

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