Layered Architecture

February 21, 2022

Richard Thomas & Brae Webb

Presented for the Software Architecture course
at the University of Queensland

THE UNIVERSITY
OF QUEENSLAND
AUSTRALIA

Layered Architecture

February 21, 2022

Richard Thomas & 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.

```
1  import gui
2  import database

4  button = gui.make_button("Click me to add to counter")
5  button.onclick(e =>
6      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 clusters of components (i.e. *layers*)

---

[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.

and restricts how components in one layer can communicate with components in another layer. A layered architecture creates superficial boundaries between the layers. Often component boundaries are not enforced by the implementation 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 another layer fulfilling the contract.

3.1 Standard Form

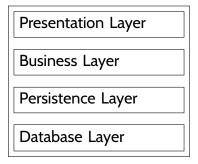| Presentation Layer |
| Business Layer |
| Persistence Layer |
| Database Layer |

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

1  import gui
2  import business

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

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

```
» cat business.code
1  import persistence

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

```
» cat persistence.code
1  import db

3  class ClickCounter:
4      clicks = 0

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

9      def get_click():
10         return clicks

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

15 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.
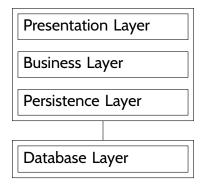
4

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 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.
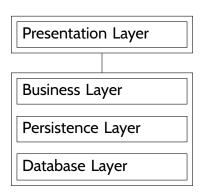

Figure 6: Traditional layered architecture with a separately deployed presentation layer.

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

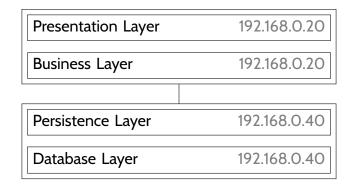presentation layer remains the same but the communication between the business and persistence layers is now via REST.[7]

Figure 7: A contrived example of a deployment variation.

```
» cat business.code

1   import http

3   def click():
4       http.post(
5           address="192.168.0.40",
6           endpoint="/click/add",
7           payload=1
8       )
```

```
» cat persistence.code

1   import db
2   import http

4   class ClickCounter:
5       ... # as above

7   click_counts = ClickCounter()

9   http.on(
10      method="post",
11      endpoint="/click/add",
12      action=(payload => click_counts.add(payload))
13  )
```

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

---

[7]https://restfulapi.net/

6

3.3   Layered Principles

Separating software into layers is intended to increase the modularity and isolation of the components within each layer. Isolation is provided by defining a public interface through which all communication with the layer is to be performed.

> **Definition 1. Layer Isolation Principle**
>
> Layers should not depend on implementation details of another layer. Layers should only communicate through well defined interfaces (*contracts*).

Layering should be enforced. One layer should not "reach across" another layer to access behaviour implemented in some other layer. For example, in our standard form of the layered architecture, if the presentation layer uses a component from the presistence layer, it defeats the intent of having a business layer in the architecture.

A consequence of this is chains of message passing. An extreme example would be if the presentation layer needed to display some information from the database, the presentation layer would send a message to the business layer to get the object to be displayed. The business layer would send a message to the persistence layer to retrieve the object. The persistence layer would then send a message to the database layer to load the object.

Typically, there would not be a need to send messages from the highest to lowest layer. If the business layer knew it had an up-to-date copy of the object, it would return it to the presentation layer without messaging the persistence layer. If the persistence layer had already retrieved the object from the database, it would return it to the business layer without messaging the database layer.

> **Definition 2. Neighbour Communication Principle**
>
> Components can communicate across layers only through directly neighbouring layers.

Layers should be hierarchical. Higher layers depend on services provided by lower layers but not vice versa. This dependency is only through a public interface, so that components in the lower layer may be replaced by another component implementing the same interface. Components in a lower layer should not use components from a higher layer, even if the layers are neighbours.

> **Definition 3. Downward Dependency Principle**
>
> Higher-level layers depend on lower layers, but lower-level layers do not depend on higher layers.

Downward dependency does not mean that data is not passed to higher layers. It does not even mean that control cannot flow from a lower level to a higher level. The restriction is on dependencies or usage, not on data or control flow. A lower layer should not use components from a higher layer, even through the higher layer's interface. Breaking this increases the overall coupling of the system and means it is no longer possible to replace a lower layer with another layer.

Lower layers need a mechanism to be able to notify a higher layer that something has happened, of which the higher layer needs to be aware. A common example of this is the presentation layer wants to be notified if data that it is displaying has been updated in a lower layer. The observer design pattern[8] is a common solution to this notification issue. The component responsible for displaying the data in the presentation layer implements the *Observer* interface. The object containing data that may be updated implements the *Subject* interface. The subject and observer interfaces are general purpose interfaces that do not belong to the presentation layer. The lower layer uses the observer interface to notify the presentation layer that data has changed and the presentation layer can decide whether to retrieve the new data

---

[8]https://refactoring.guru/design-patterns/observer

and display it. This allows the higher layer to be notified of events, without the lower layer using anything from the higher layer.

The same issue occurs with error handling and asynchronous messaging. If a component in a higher layer sends a message, through an interface, to a component in a lower layer, the component in the lower layer needs a mechanism to report errors. A simple boolean or error code return may work in some situations, but often that is not appropriate. If the message is meant to return a value, in most languages it cannot also return an error result. There may also be different types of errors that need to be communicated to the higher layer. (e.g. The call from the higher layer broke the contract specified in the interface. Or, the lower layer is encountering a transient fault and the higher layer should try again later.) Exception handling would work, if all layers are within one executable environment, but a key purpose of a layered architecture is to allow separation of the layers, so throwing an exception is not appropriate.

Callbacks[9] are used to deal with this issue for both error handling and asynchronous messaging. A component from a higher layer in the architecture passes a function as a parameter when it sends a message to a component in a lower layer. This function is called by the component in the lower layer of the architecture to report an error or to indicate that an asynchronous call has completed.

> **Definition 4. Upward Notification Principle**
>
> Lower layers communicate with higher layers using general interfaces, callbacks and/or events. Dependencies are minimised by not relying on specific details published in a higher layer's interface.

The subject and observer interfaces are examples of supporting logical infrastructure. Logging frameworks are another example of supporting infrastructure. Commonly, all layers will need to use the logging framework. These are typically defined in separate "layers" that can be used by any of the other layers. These are sometimes called *sidecar* or acquaintance layers, as visually they are often drawn on the side of the layered diagram.
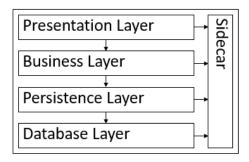

Figure 9: Layered architecture with sidecar.

> **Definition 5. Sidecar Spanning Principle**
>
> A sidecar layer contains interfaces that support complex communication between layers (e.g. design patterns like the observer pattern) or external services (e.g. a logging framework).

A purist design approach says that a sidecar layer may only contain interfaces. In some environments, an architecture may decide that multiple sidecars are beneficial, and may even use these for reusable components from broader organisational systems or for objects that hold data passed to higher layers. Figure 10 is an example of using sidecars for both of these purposes in a J2EE[10] application.

In the example shown in figure 10, the servlets and action classes layer is equivalent to the presentation layer. The controller and service classes layers are a further partitioning of the business layer. The DAO

---

[9] https://www.codefellows.org/blog/what-is-a-callback-anyway/
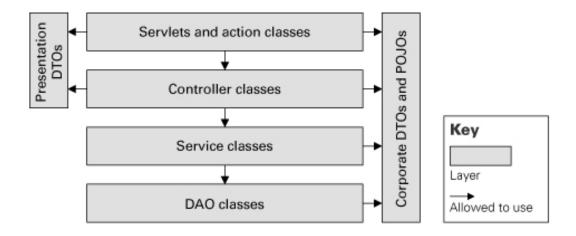[10] https://www.oracle.com/java/technologies/appmodel.html

Figure 10: Layered architecture with sidecars delivering implementation (figure 2.27 in Clements et al, 2010) [2].

(Data Access Objects) classes layer is equivalent to the persistence layer.

The Presentation DTOs (Data Transfer Objects) sidecar contains simple JavaBeans[11] that contain data that is to be displayed. This approach takes advantage of J2EE's mechanism that automatically populates and updates data in the presentation layer.

The Corporate DTOs and POJOs (Plain Old Java Objects) sidecar contains classes implemented by corporate-wide systems, and which are shared between systems. These provide common data and behaviour that spans multiple layers in many systems.

3.3.1 Closed/Open Layers

Some textbooks discuss the concept of closed and open layers. This is a way to descrive how communication flows between layers. Layers are categorised as either *open* or *closed*. By default layers are *closed*. Closed layers prevent direct communication between their adjacent layers, i.e. they enforce the neighbour communication principle. Figure 11 shows the communication channels (as arrows) in a completely closed architecture.
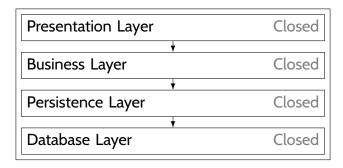

Figure 11: All layers closed requiring communication to pass through every layer.

An architecture where all layers are closed provides maximum isolation. A change to the communication contracts of any layer will require changes to at most one other layer.

Some architects will advocate that there are some situations where an *open* layer may be useful. Open layers do not require communication to pass through the layer, other layers can "reach across" the layer. The preferred approach is to use general interfaces, callbacks and/or events, as discussed in the sections describing the downward dependency, upward notification, and sidecar spanning principles. This provides mechanisms that allow data and control to flow both up and down in a layered architecture, without

---

[11]https://www.educative.io/edpresso/why-use-javabean

breaking the isolation principle that was the original intent of using a layered architecture. Open layers in architecture design should be avoided.
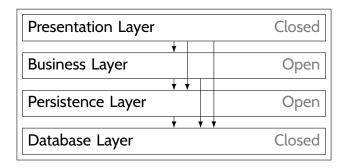

Figure 12: A wolf in layer's clothing [3].

3.4   Advantages

The layer isolation principle means that the implementation of a layer can be changed without affecting any other layer, as long as the interface does not change.

The layer isolation principle also means that a developer only needs to understand the public interface to use a layer, and not its implementation details.

The neighbour communication and downward dependency principles mean that if a layer changes its public interface, at most one other layer needs to change.

The upward notification and sidecar spanning principles mean that complex systems, with sophisticated flows of control and data, can be implemented while maintaining the other layered architecture design principles.

Lower layers in the architecture can be designed to deliver common services that may be reused across multiple applications. (e.g. The persistence layer can be designed to allow general purpose access to the database layer, allowing any type of database to be substituted into the system.)

Layers may be deployed on different computing infrastructure. This enables the hardware to be optimised for the types of services provided by just one layer. It also enables scaling and replication by allowing layers to be duplicated across multiple servers.

3.5   Disadvantages

Poorly designed layers will encourage developers to break the layered architecture design principles in order to get the system to work. This can lead to a system that in detail more closely resembles a big ball of mud, than a layered design.

Layering often introduces performance penalties. Requiring a chain of message passing to obtain a service from a much lower layer in the architecture adds to the cost of delivering the behaviour.

References

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

[2] D. Garlan, F. Bachmann, J. Ivers, J. Stafford, L. Bass, P. Clements, and P. Merson, *Documenting Software Architectures: Views and Beyond*. Addison-Wesley Professional, 2nd ed., 2010.

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