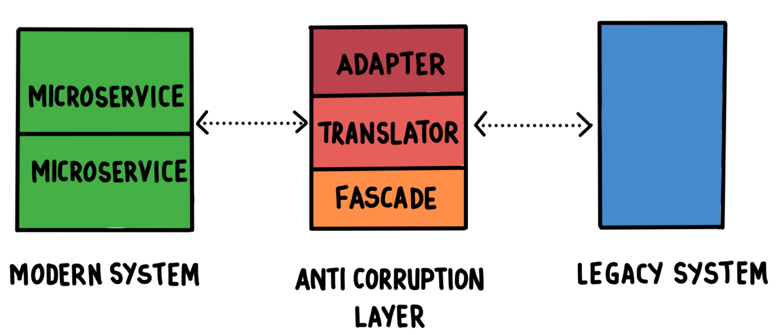
**Anti Corruption Layer (ACL)**

An Anti-Corruption Layer (ACL) is a set of patterns placed between the domain model and other bounded contexts or third party dependencies.

**What is Anti-Corruption-layer**

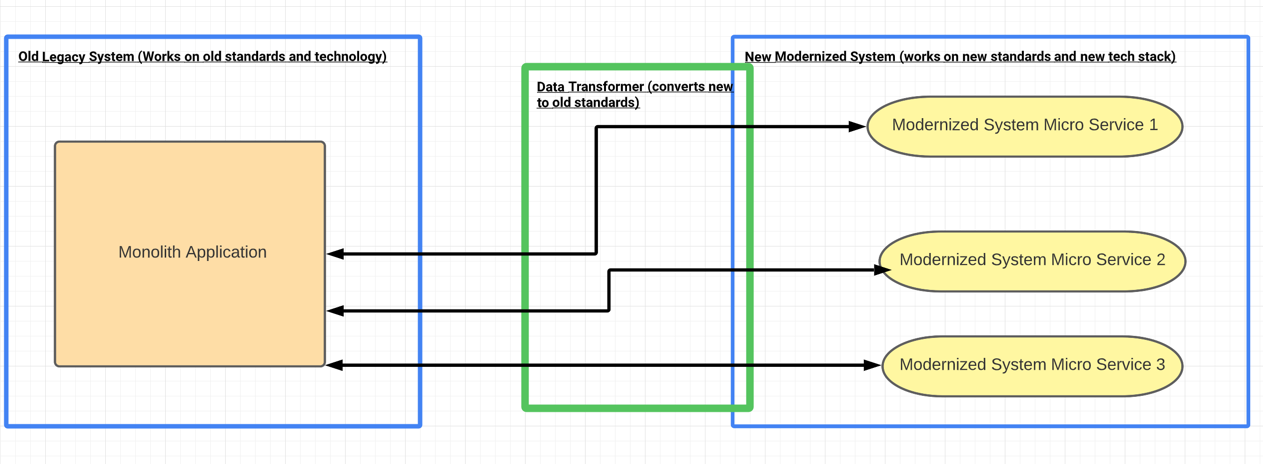
Implement a façade or adapter layer between different subsystems that don't share the same semantics. This layer translates requests that one subsystem makes to the other subsystem. Use this pattern to ensure that an application's design is not limited by dependencies on outside subsystems.

* Anti Corruption Layer, a concept discussed first in Domain Driven Development, adds a layer that would help in protecting the new system from corruption from legacy applications. These are also useful when required to maintain the sanity of your layer when there are changes to external systems it depends on. The Anti Corruption Layer could be uni/bi-directional depending on the needs of the system.
* The Anti Corruption Layer uses a combination of **Facade, Adapter, and Translators** to isolate the system. It helps in simplifying the interfaces from the legacy system and providing a simplified, yet improved interface integration with the newer system. The translators could help in mapping incompatible models with each other.

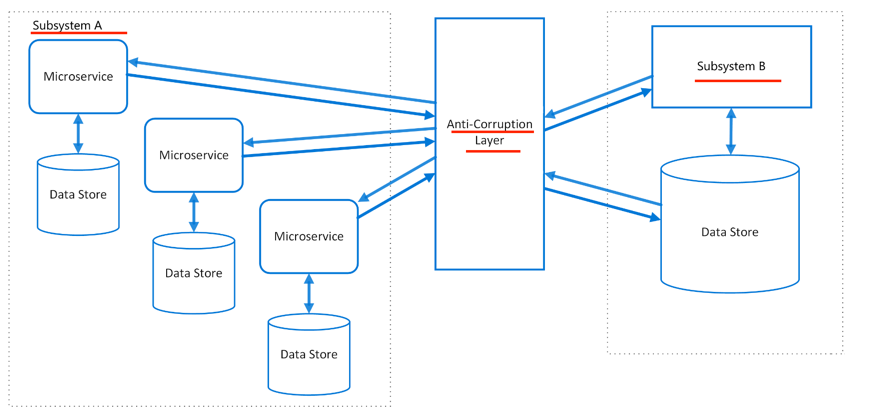


**What is the Context, What is the Problem?**

* Most applications rely on other systems for some data or functionality. For example, when a legacy application is migrated to a modern system, it may still need existing legacy resources. New features must be able to call the legacy system. This is especially true of gradual migrations, where different features of a larger application are moved to a modern system over time.
* Often these legacy systems suffer from quality issues such as convoluted data schemas or obsolete APIs. The features and technologies used in legacy systems can vary widely from more modern systems. To interoperate with the legacy system, the new application may need to support outdated infrastructure, protocols, data models, APIs, or other features that you wouldn't otherwise put into a modern application.
* Maintaining access between new and legacy systems can force the new system to adhere to at least some of the legacy system's APIs or other semantics. When these legacy features have quality issues, supporting them "corrupts" what might otherwise be a cleanly designed modern application.
* Similar issues can arise with any external system that your development team doesn't control, not just legacy systems.



**So.. what is the solution ?? Its ACL**

Isolate the different subsystems by placing an anti-corruption layer between them. This layer translates communications between the two systems, allowing one system to remain unchanged while the other can avoid compromising its design and technological approach. 

* ACL does not have to process only the "old application → new microservice" communication. It can also support the communication in the opposite direction. Let's assume that after processing its tasks, your microservice must return the results somewhere. The old application would e.g. save them in a database (the global database that keeps all the data necessary for the monolithic application), where they would be required by another functionality. Currently, there is no other microservice that would need this data and the only available method is to save the data within the old structure. And here comes ACL again. You become its client and it will process data saved in a new format submitted using a brand new protocol (e.g. HTTP and REST architectural pattern). Therefore, ACL can act as a wrapper (proxy) that changes the data transmission method (e.g. it will receive data via REST to save it directly to a database). However, it can also act as an adapter that translates data in the new format to the old format, recognized by the old application.
* The diagram above shows an application with two subsystems. Subsystem A calls to subsystem B through an anti-corruption layer. Communication between subsystem A and the anti-corruption layer always uses the data model and architecture of subsystem A. Calls from the anti-corruption layer to subsystem B conform to that subsystem's data model or methods. The anti-corruption layer contains all of the logic necessary to translate between the two systems. The layer can be implemented as a component within the application or as an independent service.

**So ACL will solve all the Problems**

Well ACL is a strong migration pattern but there are some issues and considerations like :

* The anti-corruption layer may add latency to calls made between the two systems.
* The anti-corruption layer adds an additional service that must be managed and maintained.
* Consider how your anti-corruption layer will scale.
* Consider whether you need more than one anti-corruption layer. You may want to decompose functionality into multiple services using different technologies or languages, or there may be other reasons to partition the anti-corruption layer.
* Consider how the anti-corruption layer will be managed in relation with your other applications or services. How will it be integrated into your monitoring, release, and configuration processes?
* Make sure transaction and data consistency are maintained and can be monitored.
* Consider whether the anti-corruption layer needs to handle all communication between different subsystems, or just a subset of features.
* If the anti-corruption layer is part of an application migration strategy, consider whether it will be permanent, or will be retired after all legacy functionality has been migrated.

**So I can use ACL all the time?**

**Use ACL pattern when** :

* A migration is planned to happen over multiple stages, but integration between new and legacy systems needs to be maintained.
* Two or more subsystems have different semantics, but still need to communicate.
* When functionalities needs to be exposed to external systems through standard interface.

**Don't use ACL pattern when** :

* There are no significant semantic differences between new and legacy systems.