# Accessing complex back-end logic from the presentation layer

Imagine that you are dealing with a whole range of complex classes, all of which you would need to access from a single layer of your application. It could be, for example, code that has been auto-generated from a WSDL definition. Or it could be some manually written code where different classes retrieve data from different data sources.

This scenario is very typical, as it uses the commonly-used multi-layer architecture. In such architecture, there would be a separate layer responsible for presentation, a separate layer responsible for back-end business logic, and a separate layer responsible for data storage. But in real-life applications, it often isn't as simple as this. The application may be retrieving its data from multiple sources, such as several different database types and external services.

Because you will need to access all of these classes from the same layer within your application, using those classes directly would probably not be the most optimal thing to do. You would need to refer to all of these classes from the other system components that need to access them.

This is especially problematic with auto-generated classes, as you won't be able to easily create abstractions for them. So, you will either have to manually edit auto-generated code (which isn't a good idea) or pass the concrete classes as references to the components that need them (which is also a bad idea and a clear violation of the dependency inversion principle).

The problem will become even more apparent if those complex classes are meant to be updated fairly frequently. In this case, you will have to keep updating all the references to them.

Another problem associated with complex logic is operations that are expensive to run. But what if you need to access the results of such operations frequently? Well, luckily, there are design patterns that can help you solve this problem.

## Suitable design patterns

### Facade

Facade is a class that controls access to a set of complex objects and makes it simple. Most often, just like an Adapter, it would change the access interface into these classes. However, unlike Adapter, it will usually be a wrapper around several of such classes. Or it may be responsible for a diverse set of interactions between a number of moving system parts rather than just converting one type of a call to another type.

Direct interaction with those complex classes happens only inside the Facade class, so the other system components are completely shielded from it. All the client class will be concerned about is calling the specific methods on the Facade. The client doesn't care how Facade delivers what it needs. It only cares that Facade does the job that is expected of it.

When any of the above-mentioned components need to be updated, the update in the logic will only need to happen inside the Facade, which will prevent the other parts of the system from having bugs unintentionally introduced into them by forgetting to update the logic.

### Why would you use Facade

- A convenient way of simplifying access to a complex subset of the system.
- All other application components are shielded away from having to interact with complex logic.
- Since all the code that interacts with a complex subsystem is located in one place, it's easier not to miss updates to the logic if any of the subsystem components get updated.

## Proxy

Proxy is not suitable for wrapping complex logic into a simple accessible interface, but it's still suitable for dealing with subsystem components that are not easy to work with directly.

For example, you may be in a situation where you would only need data from a particular service on rare occasions. If this data takes a long time to obtain and rarely changes, a proxy can be used to access this data once and store it in memory until it changes.

Or you may have a situation where you would need to restrict access to a particular service based either on a specific outcome of business logic or the roles that the user is assigned to. In this case, the proxy would conduct this check before the actual service is accessed.

So, essentially, a Proxy is nothing other than a wrapper around a class that has exactly the same access interface as the original class, so both classes are interchangeable. The Proxy class is there to restrict access to the original class. It can also be used to implement any pre-processing of the request if it's needed before the original class can be accessed. A Proxy can implement additional access rules that cannot be added to the original class directly.

## Why would you use Proxy

- You can abstract away all complex implementation details of accessing a particular class.
- You can apply additional request validation before you access a particular class.
- You can get it only to access the actual class when it’s necessary, which would positively affect the performance.