Problemas Comunes de Desarrollo

Compartir el modelo de la base de datos a lo largo de varios límites

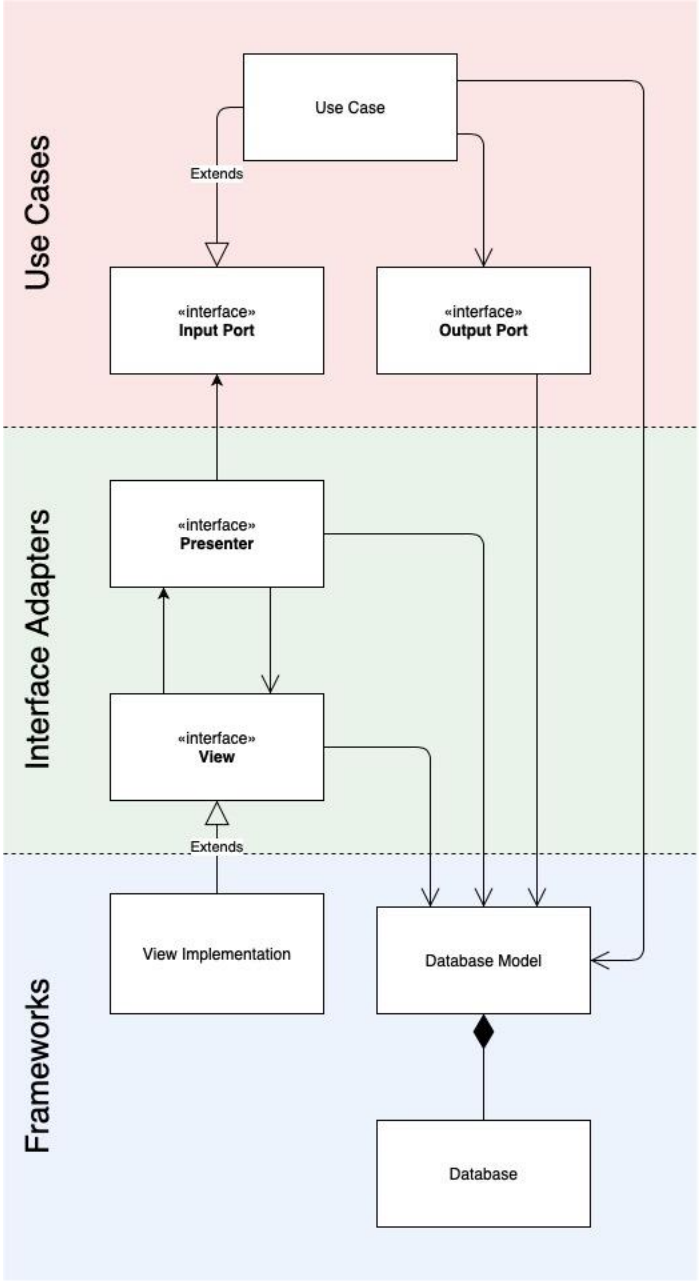

Figure 1: shared_database_model_across_boundaries.jpg

Compartir el modelo de la base de datos entre distintos límites no es una buena práctica, ya que crea una fuerte dependencia entre los componentes de la aplicación y la base de datos que estemos utilizando. Por ejemplo, si fuera necesario cambiar la base de datos de una local a una base de datos en la nube, el costo de cambiarla sería demasiado alto,

ya que habría que modificar cada uno de los componentes que dependan de esta. Además de que viola la regla de la dependencia de The Clean Architecture.

Compartir el mismo input/output data entre distintos casos de uso

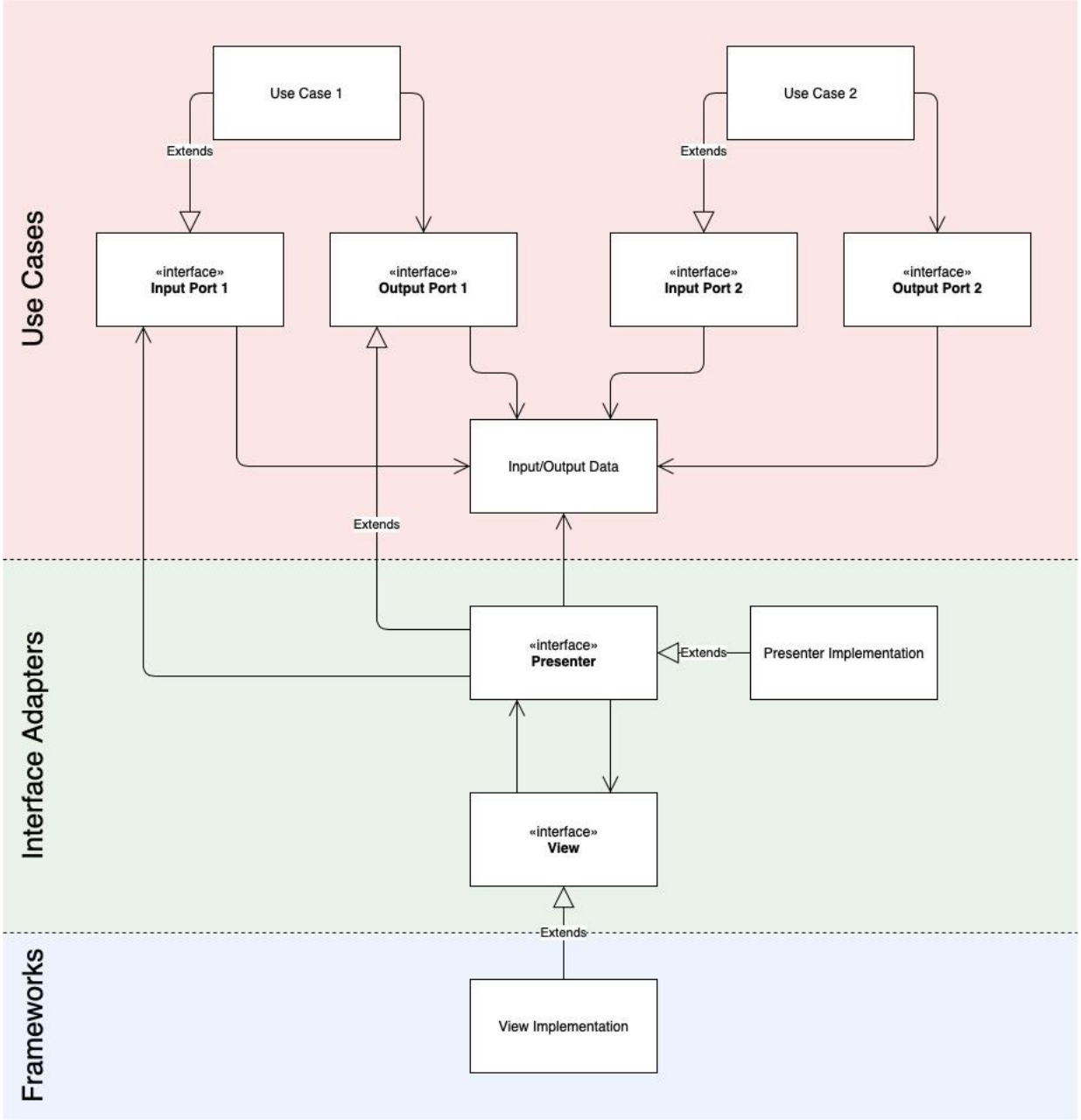

Figure 2: shared_input_output_data.jpg

Es común emplear una misma estructura de datos para pasar información entre los distintos límites de nuestra aplicación como si de un Data Transfer Object se tratara, pero esta práctica puede acarrear varios problemas, entre ellos se encuentran los conflictos que se pueden producir si más de una persona modifica la misma estructura de datos. Otro problema que se puede presentar es que aumenta la complejidad del entendimiento

del caso de uso al incorporar campos (campos, propiedades, variables de instancia, etc.) que no son utilizados por el caso de uso.

Una arquitectura limpia

La arquitectura limpia propuesta a continuación se basa en la teoría descrita por Robert C. Martin en su libro **Clean Architecture – A Craftsman's Guide to Software Structure and Design**, y por la teoría descrita por Tom Hombergs en su libro **Get Your Hands Dirty on Clean Architecture**.

La arquitectura limpia, es un tipo de arquitectura de software que permite la separación de las reglas de negocio implementadas en la aplicación y de los distintos frameworks que esta podría utilizar, lo que establece una independencia de la aplicación hacia estos. Por lo tanto la incorporación de un nuevo framework o la sustitución de otro, no afecta a las reglas de negocio implementadas por la aplicación.

La arquitectura limpia propuesta a continuación, permite separar las reglas de negocio de la aplicación ScotiaMóvil y de cualquier otra aplicación, sea móvil, sea middleware, o sea front end.

Una arquitectura limpia simple

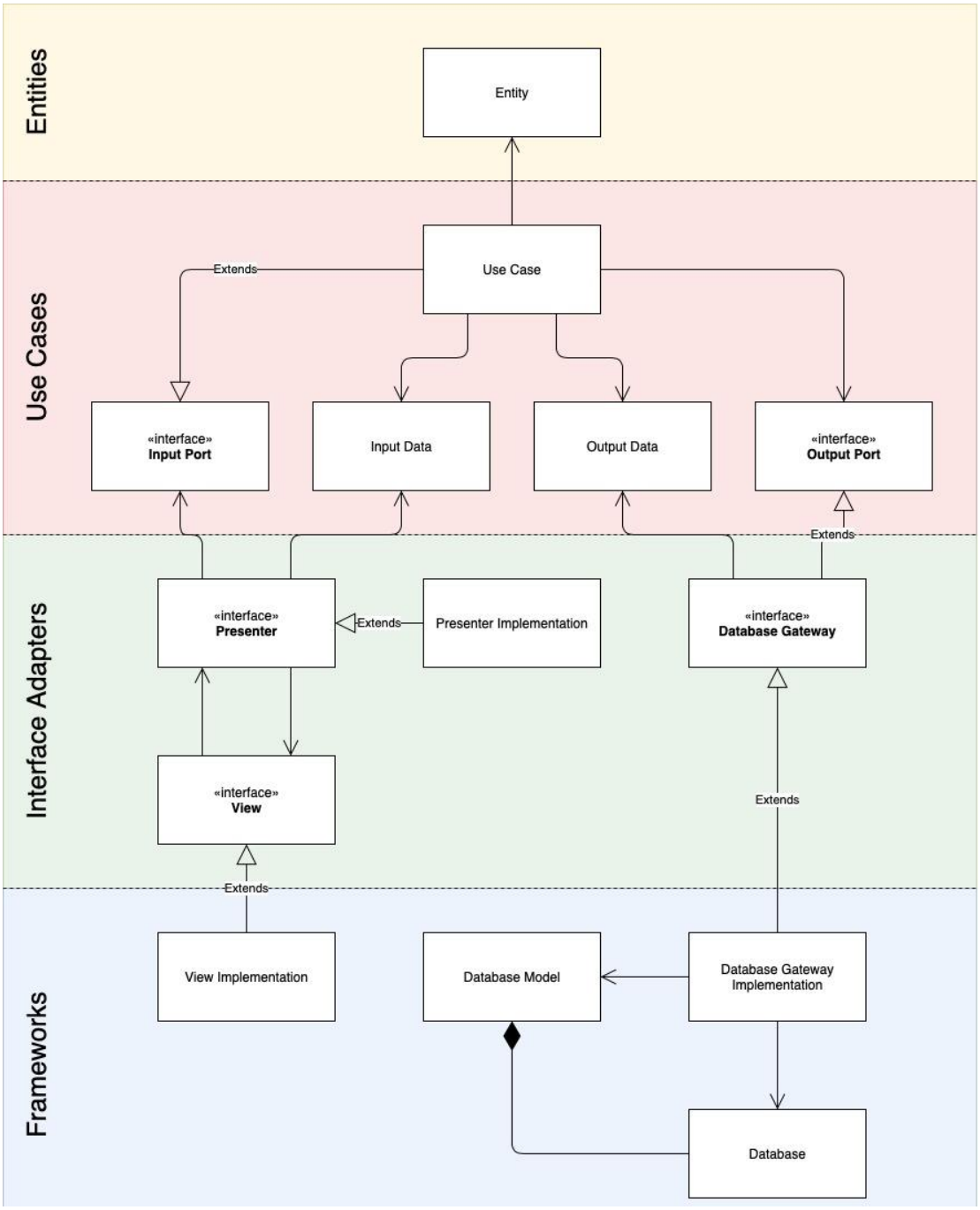

Figure 3: a-simple-clean-architecture-class-diagram.jpg

En el diagrama de clases anterior se muestra una arquitectura limpia simple, sus elementos y su comportamiento se estará describiendo a continuación.

En el diagrama anterior se muestran 4 límites, los cuales son Entities, Use Cases, Interface Adapters y Frameworks.

En el límite de Entities se encuentran las entidades de la aplicación las cuales se encuentran compuestas por información y por comportamiento. Estas implementan las Enterprise Business Rules sólo en caso de que se implemente un Rich Domain Model.

En el límite de Use Cases, se encuentran los casos de uso, los cuales implementan los Application Business Rules, y los Enterprise Business Rules sólo en caso de implementar un Anemic Domain Model. En este límite también se encuentran los Input/Output Ports, los cuales son interfaces que son utilizadas por los Interface Adapters para comunicarse con los Use Cases y por los Use Cases para comunicarse con los Interface Adapters utilizando el patrón de diseño Dependency Inversion. También se encuentran los Input/Output Data, las cuales son estructuras de datos que son utilizadas para transportar la información que es utilizada por los Use Cases y por los Interface Adapters a manera de Data Transfer Objects. Los Input/Output Data pueden ser estructuras de datos o pueden ser parámetros de un método de acuerdo a la teoría de The Clean Architecture. Por convención y por elegancia recomiendo utilizar estructuras de datos para modelar de manera más apropiada la información requerida por los Use Cases.

Los Interface Adapters son clases e interfaces que utilizan/implementan las interfaces (Input/Output Ports) expuestas por el límite de los Use Cases. Entre los componentes de este límite se encuentran el Presenter y el View los cuales describen el comportamiento que debe tener el presentador y la vista respectivamente. En este límite también se pueden encontrar componentes, por ejemplo, los gateways, los cuales describen una serie de métodos que son implementados en el límite de Frameworks para poder acceder a una base de datos o consumir una serie de microservicios.

En el límite de los Frameworks se encontrarán las implementaciones de las interfaces expuestas en el límite de los Interface Adapters, por ejemplo, la implementación de la vista, la cuál será implementada por un Activity/Fragment en Android o un UIViewController en iOS. De igual forma en este límite estarán las implementaciones de los gateways, por ejemplo, un gateway que implemente el acceso a una base de datos.

Llamar a un caso de uso sin un input port definido

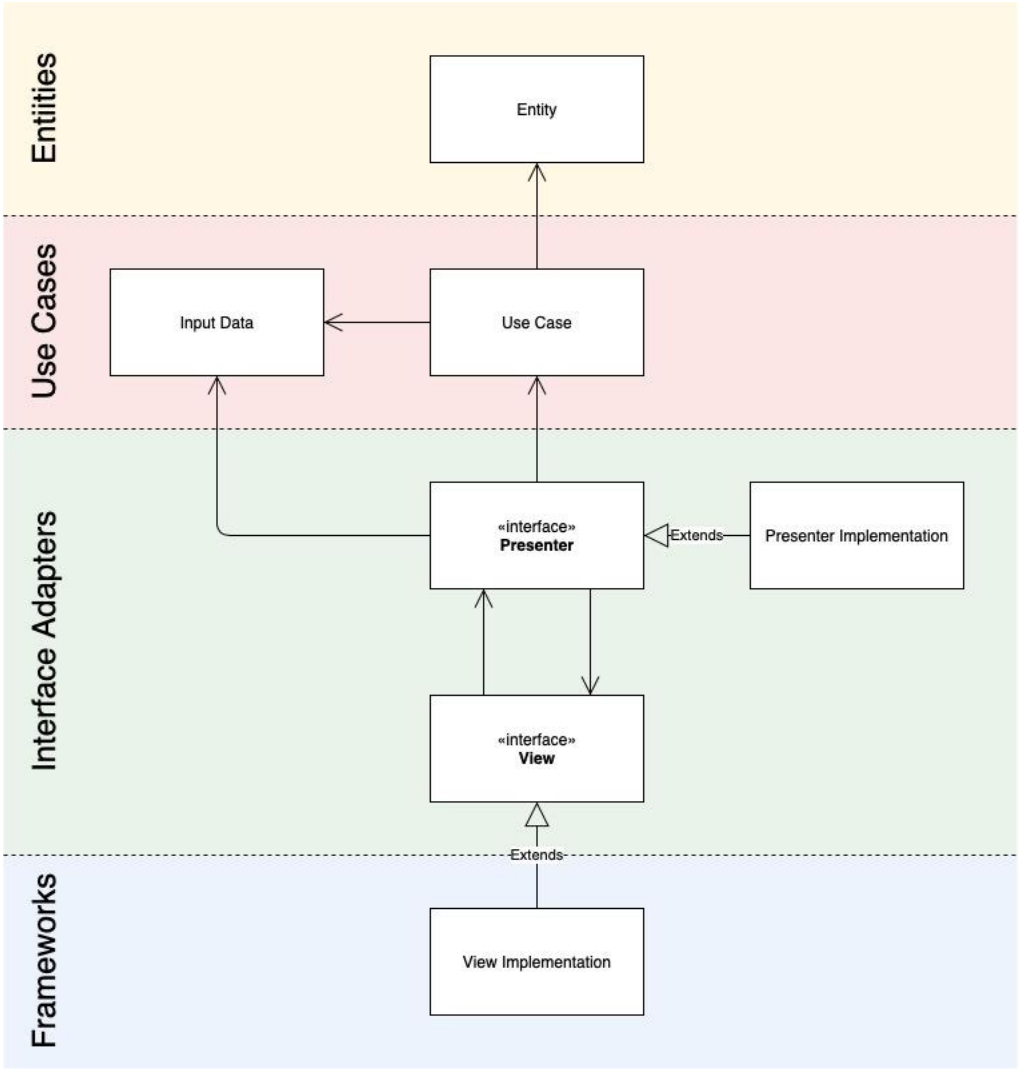

Figure 4: no-input-port.jpg

En la figura anterior se muestra un límite de casos de uso en el cual no se define un input port para el use case. El uso de esta práctica a pesar de que no afecta a la aplicación, es visto como una mala práctica de diseño, ya que establece una relación fuerte entre el presenter y el use case.

Utilizar un gateway en lugar de un output port

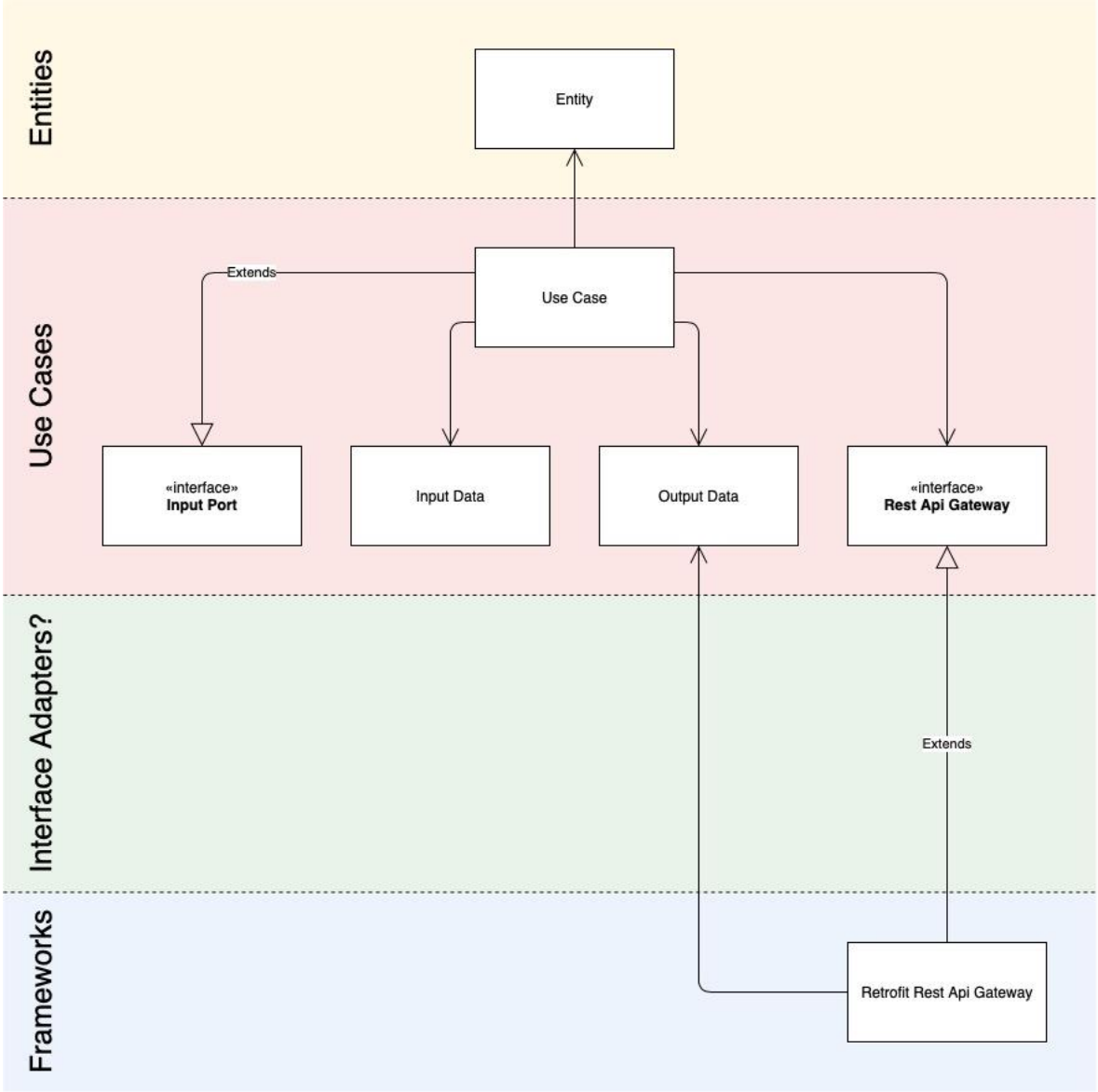

Figure 5: non-specific_output_port.jpg

En la figura de arriba se muestra a un use case haciendo uso de un gateway particular en lugar de definir un output port específico. Lo anterior también es visto como una mala práctica de diseño, ya que establece una relación fuerte entre el use case y el gateway, haciéndole ver al caso de uso todos los métodos expuestos por el gateway, los cuales no son necesarios para el use case. Para evitar esta mala práctica de diseño, se emplea el patrón de diseño Interface Segregation.

Un caso de uso simple

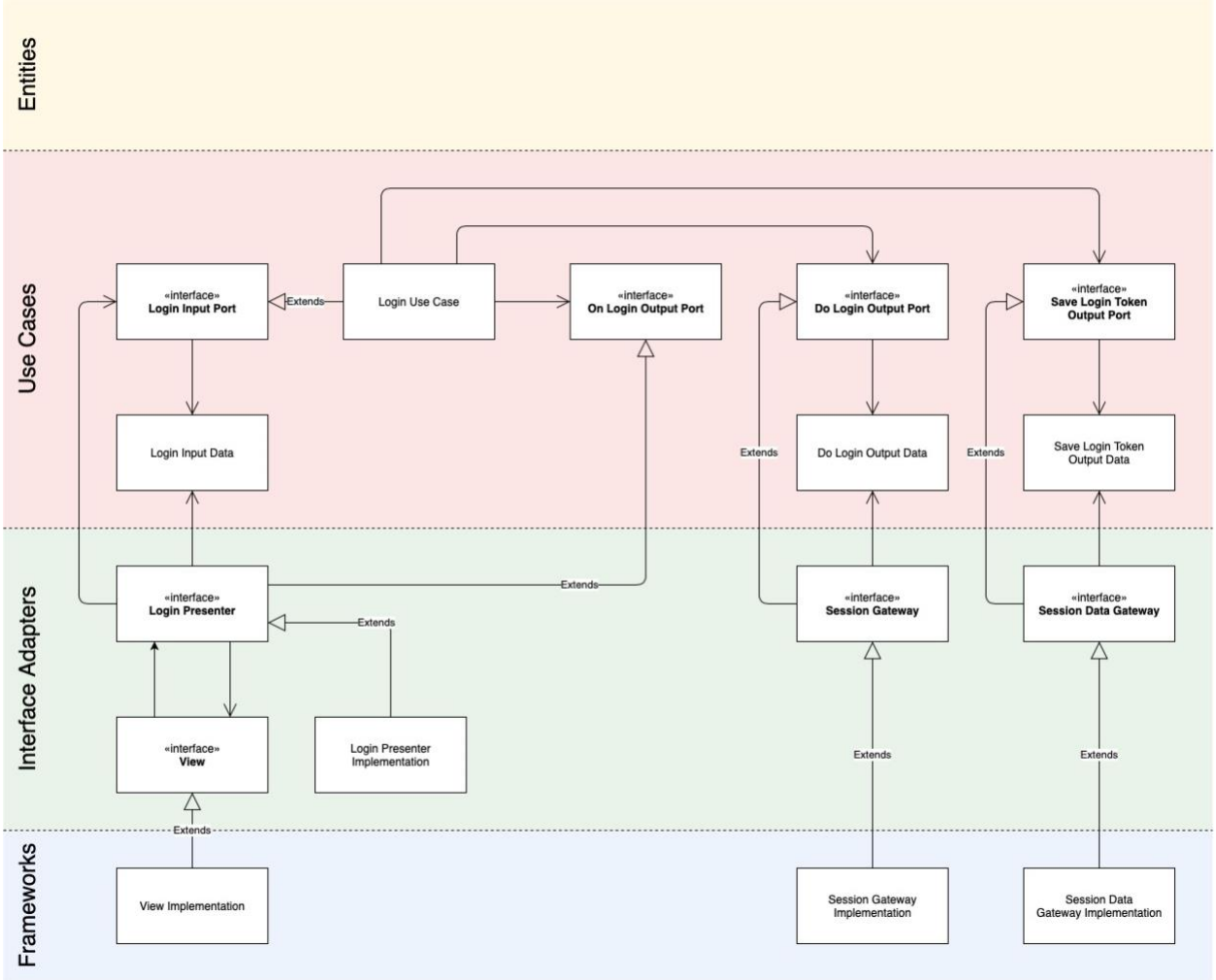

Figure 6: simple-login-use-case-class-diagram.jpg

En la figura anterior se muestra el diagrama de clases de un caso de uso para iniciar sesión en algún sistema de cómputo, el cual requiere mandar a llamar a un servicio para iniciar sesión y almacenar el token obtenido de dicho servicio, el cual será usado para poder llamar a otros servicios después, en una base de datos.

A continuación, se describirán cada uno de los elementos y su comportamiento.

En el límite de los Use Cases, se encuentra el Login Use Case el cual se encargará de llamar al servicio para iniciar sesión a través del Do Login Output Port, una vez obtenido el token del servicio, este será guardado a través del Save Login Token Output Port. Por otro lado, el Login Input Port, es implementado por el Login Use Case para poder ser llamado por los Interface Adapters. Una vez que el caso de uso haya termido de guardar el token, este usará el On Login Output Port, para comunicarle a un componente en los Interface Adapters que el inicio de sesión se llevó a cabo. El Login Input Data, Do Login

Output Data y Save Login Token Output Data, son estructuras de datos utilizadas para transportar la información requerida por los Use Cases y los Interface Adapters.

En el límite de los Interface Adapters se encuentran el Presenter y el View, los cuales son interfaces que describen el comportamiento que debe tener el presentador y la vista respectivamente. El Presenter utiliza el Login Input Port para comunicarse con el Login Use Case, de igual forma implementa la interfaz On Login Output Port para que el Login Use Case le pueda comunicar que el inicio de sesión se llevó a cabo. El Session Gateway y el Session Data Gateway son interfaces que implementan los output ports del Login Use Case, de esta manera se pueden segregar las interfaces que después serán implementadas

En el límite de los Frameworks se encuentran el View Implementation la cual puede ser una Activity o un Fragments en el caso de Android. Se encuentran el Session Gateway Implementation y el Session Data Gateway Implementation, los cuales pueden ser implementados por Retrofit/Volley y Room/Realm en el caso de Android.

Casos de uso sin input/output data

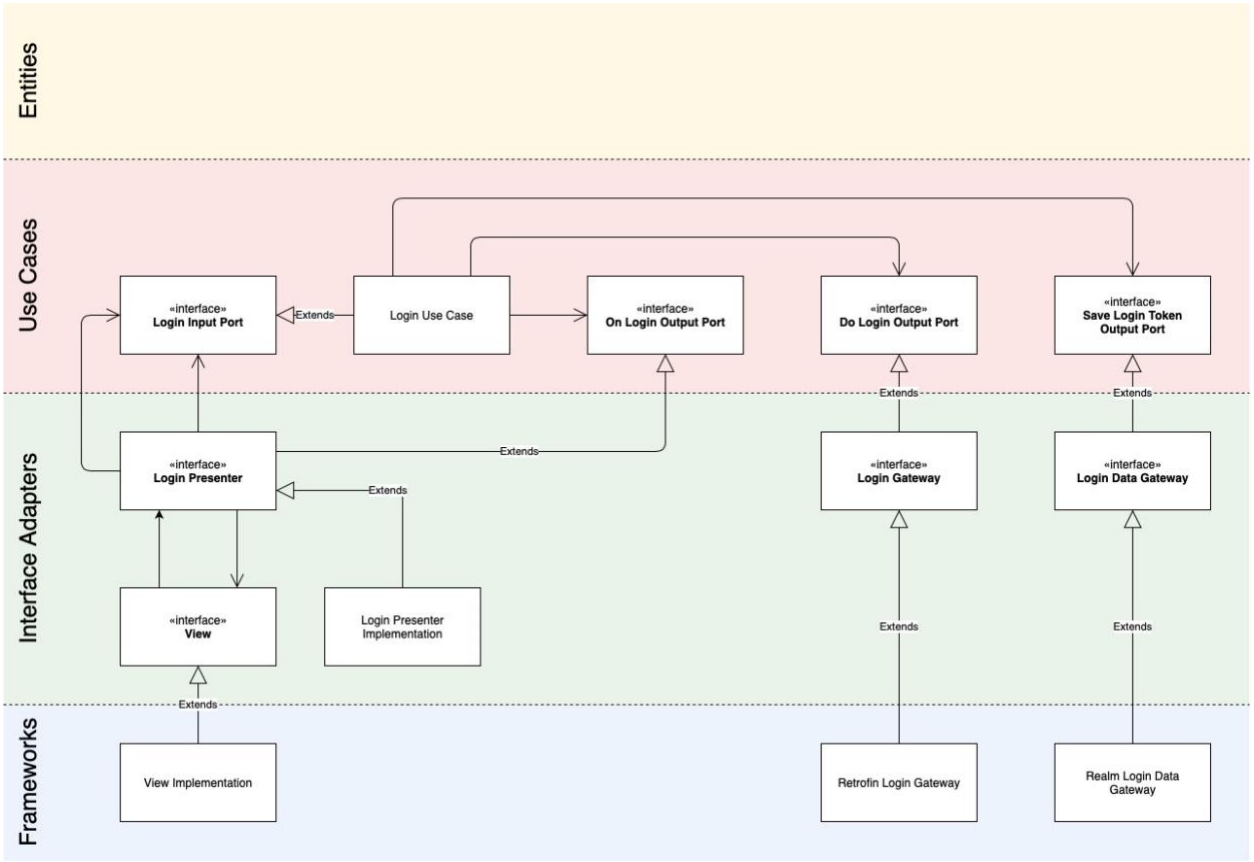

Figure 7: simple-login-use-case-without-input-output-data-class-diagram.jpg

En la figura anterior fueron omitidos los input/output data, y aunque la teoría de The Clean Architecture establece que la información puede ser proporcionada a través de parámetros en los métodos, el uso de estructuras de datos es más apropiado.

El límite de utilidades

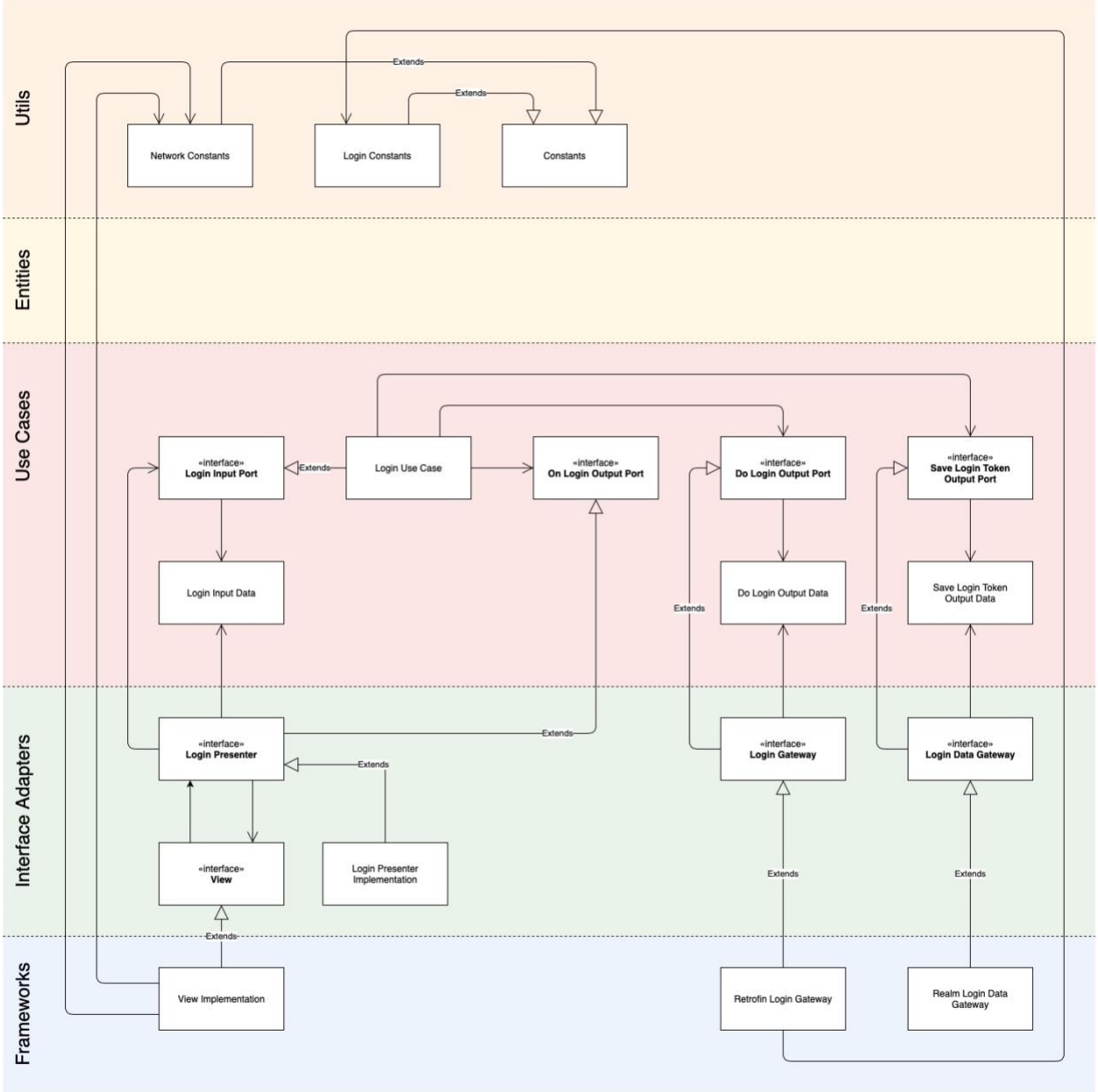

Figure 8: simple-login-use-case-with-utils-boundary-class-diagram.jpg

A veces es necesario crear algunas clases de utilidades para poder aplicar ciertos cambios a cadenas de texto, aplicar formatos específicos a cantidades de dinero o incluso definir constantes. En el límite de Utils deberán encontrarse todas esas clases que necesitemos de ayuda.

Es común que los programadores declaren TODAS las constantes de la aplicación en un solo archivo, lo cual incrementa la posibilidad de que se presente algún conflicto al editar

el archivo, es por eso que, en el diagrama anterior, se establece una separación de las constantes a través de una jerarquía de clases.

El límite de servicios

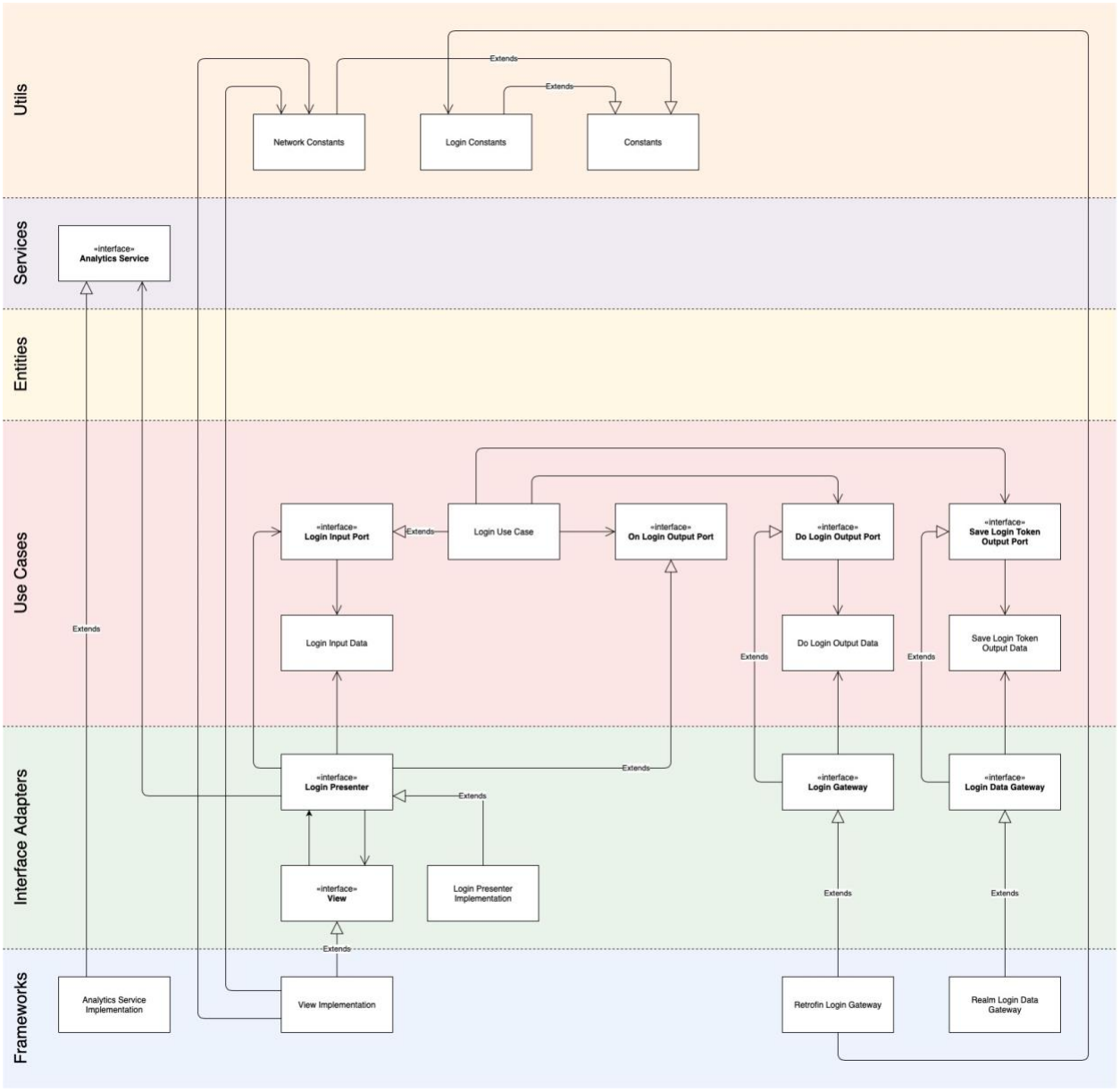

Figure 9: simple-login-use-case-with-services-boundary-class-diagram.jpg

En el caso de la aplicación de ScotiaMóvil, el uso de algunos servicios, por ejemplo, Firebase, puede ser incorporado utilizando el patrón de diseño Dependency Inversion e incorporando un nuevo límite llamado Services. En este límite estarán alojadas interfaces que después serán implementadas en el límite de Frameworks.

El principio de la segregación de interfaces

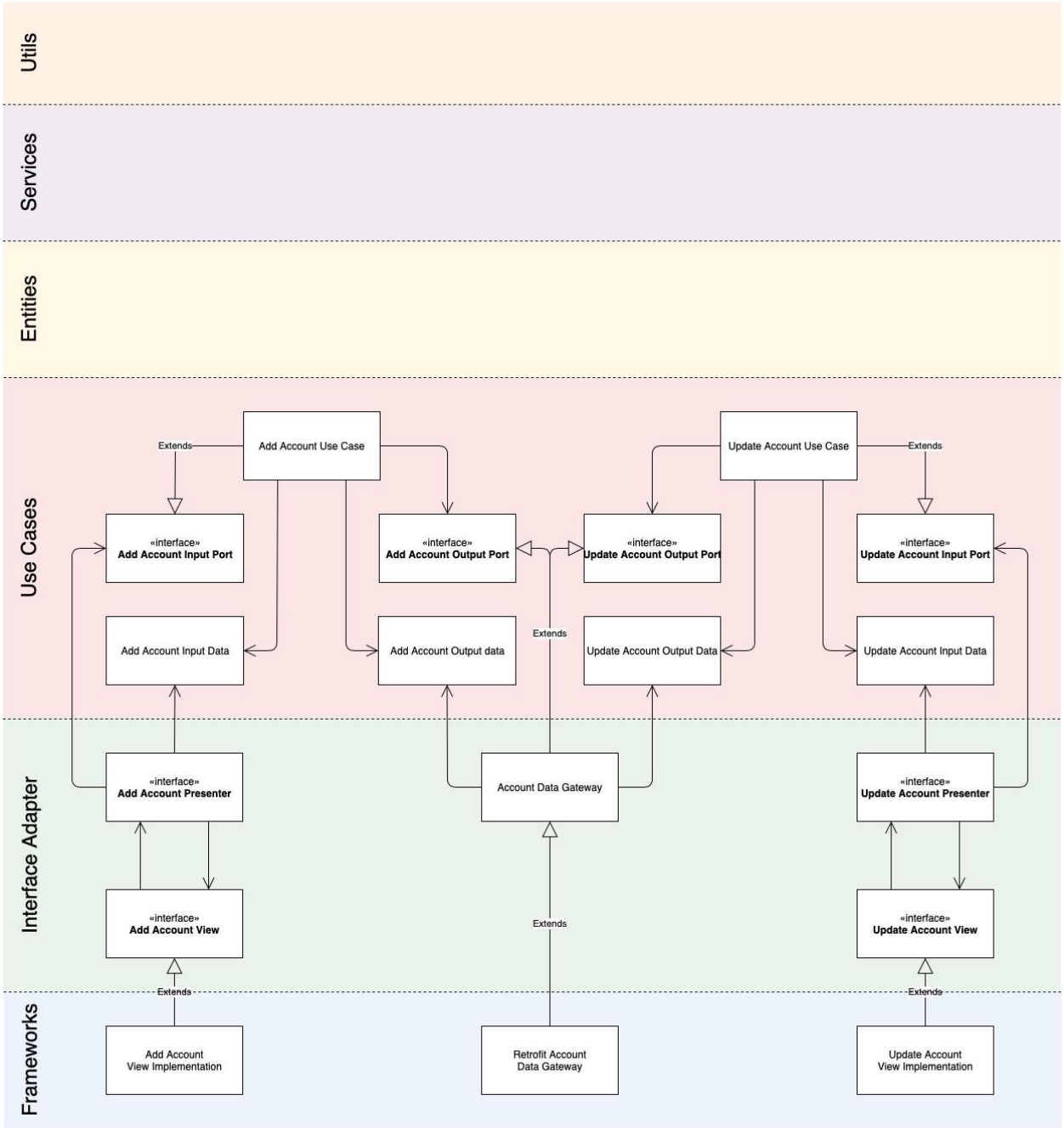

Figure 10: interface-segregation-principle.jpg

El principio de la segregación de interfaces, establece que un componente sólo debe de poder ver aquellas operaciones (métodos, funciones, etc.) que necesita de otro componente, por lo tanto, en lugar de que un componente A, vea todas las operaciones de un componente B, las operaciones que el componente A necesita del componente B, se segregan en una interfaz, que después será implementada por el componente B, de

esta manera se logra que la dependencia que existe entre componentes, este limitada únicamente por las operaciones que necesitan.

En el diagrama de clases de arriba se muestra un ejemplo del principio de la segregación de interfaces, en el cual las operaciones del Account Data Gateway, son segregadas en interfaces, en este caso, en los output ports de los use cases.

Una arquitectura en capas

Una de las dudas que surgió durante la propuesta de la arquitectura para la aplicación de ScotiaMóvil, es si The Clean Arquitecture era necesaria para la aplicación, por lo que en caso de que fuera excluida de la propuesta, se planteó una segunda arquitectura basada en una arquitectura en capas como se muestra más abajo.

Una arquitectura en capas

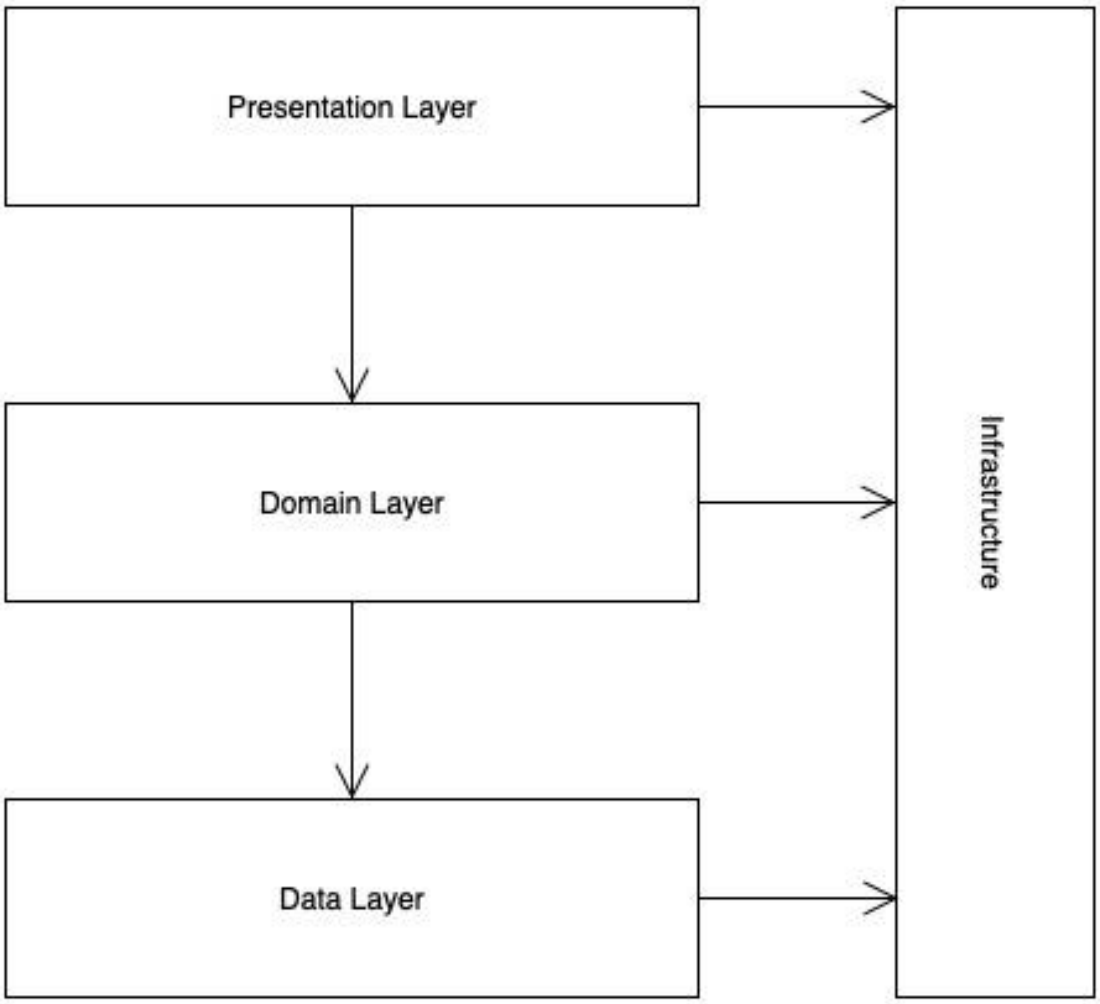

Figure 11: n-tier-architecture.jpg

La arquitectura en capas, es un tipo de arquitectura de software que se puede componer de n-capas dependiendo de las necesidades de cada aplicación. En el caso de la aplicción de Scotiamóvil, se propuso una arquitectura de 4 capas. La Presentation Layer, la cual contendrá la lógica de presentación de la aplicación, la Domain Layer, la cual contendrá todas las reglas de negocio, la Data Layer, la cual contendrá el acceso a base

de datos y microservicios, y por último la Infraestructure Layer, la cual contendrá utilidades, servicios y otros frameworks.

A continuación, se mostrará una subdivisión de las capas, de tal forma que podemos utilizar algunas de las características y teorías de The Clean Architecture en el proyecto, estableciendo así, una arquitectura más sólida, mantenible, y escalable.

Una arquitectura en capas subdividida

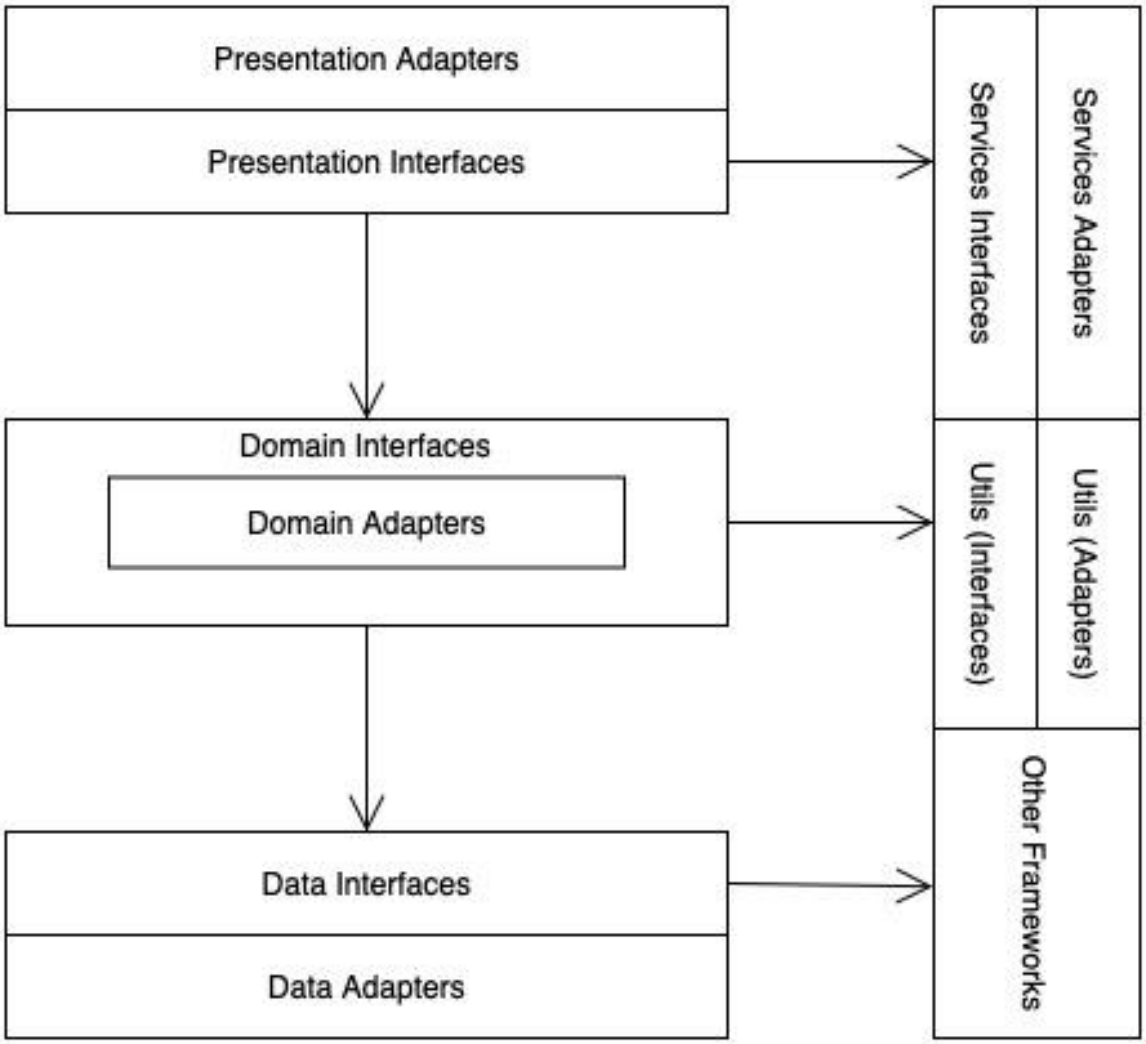

Figure 12: n-tier-architecture-subdivided.jpg

En el diagrama anterior se muestra la subdivisión propuesta para la arquitectura en capas. Cada una de las subcapas serán descritas a continuación.

En la Presentation Layer, se encuentra la subcapa Presentation Interfaces, las cuales son interfaces que describen el comportamiento de la vista de la aplicación, mientras que la subcapa Presentation Adapters, son clases que implementan las interfaces definidas en la subcapa anteriormente mencionada, estas implementaciones incluyen, la implementación del presentador, la implementación de la vista (Activity/Fragment/UIViewController), etc.

En la Domain Layer, se encuentran la subcapa Domain Interfaces, la cual son interfaces que describen el comportamiento de los casos de uso de la aplicación y que, a su vez, dependen de las interfaces en la subcapa Data Interfaces que se encuentra en la Data Layer. Mientras que la subcapa Domain Adapters, está aislada del resto de la arquitectura, de esta manera se consigue la independencia de las reglas de negocio similar a The Clean Architecture.

En la Data Layer, se encuentran la subcapa Data Interfaces, las cuales describen el comportamiento de los componentes que se encargaran de consultar la base de datos o de consumir algún microservicio. La subcapa Data Adapters, está aislada de la Domain Layer, de esta manera se puede crear una independencia de los frameworks empleados en esta subcapa, similar al límite Frameworks en The Clean Architecture.

Y por último tenemos la Infraestructure Layer, la cual está subdividida en las siguientes capas. La subcapa de servicios, la cual está conformada por Services Interfaces y Services Adapters, permite la independencia de algunos frameworks, por ejemplo, Firebase. La subcapa de utilidades, permite la independencia de algunas características del SO o Framework, por ejemplo, manipular imágenes (Picasso) o acceder a algún archivo almacenado en el dispositivo (Resources/File System APIs). Y por último Other Frameworks, esta subcapa permite la incorporación de algunos frameworks, los cuales difícilmente se pueden abstraer a través del patrón de diseño Dependency Inversion y que por cuestiones técnicas (que es imprescindible), se propuso para el proyecto, por ejemplo, RxJava o Dagger.