# Sistema Clínico

En esta práctica se ha realizado una prueba de concepto diseñando una posible implementación de un sistema clínico íntegramente sobre la blockchain de Ethereum. El desarrollo se ha hecho siguienedo el principio de TDD y para conseguir la versión final se han realizado varias iteraciones que paso a describir a continuación:

## Iteración 1

En esta primera iteración se plantearon dos componentes:

* SistemaClinico.sol: Contenía toda la información del sistema así como la lógica de negocio y puntos de entrada para interactuar con él. Distinguía entre 3 tipos de actores: owner, médicos y pacientes.

El owner correspondía con la dirección que había desplegado el contrato y era el único que podía ejercer labores administrativas, como por ejemplo añadir médicos o borrarlos.

El médico podía interactuar con el sistema para realizar su trabajo, es decir, dar de alta pacientes, consultar su expediente, añadir/modificar/terminar tratamientos.

El paciente realmente no tiene interactuación directa sobre el sistema, es simplemente un actor implicado en el proceso, pero que es manejado por otros.

* Expediente.sol : Contenía la información del expediente de un paciente. Es decir, para cada paciente que se daba de alta en el sistema, se desplegaba un nuevo contrato de este tipo para almacenar en el todo el historial médico del mencionado paciente.

Tras acabar esta iteración se analiza el resultado y se observa lo siguiente:

* Hay una muy baja cohesión, ya que toda la lógica del sistema está contenida en un único contrato.
* Además de esto, tenemos acoplada la lógica de negocio del sistema a los datos del mismo, de modo que ante un bug en dicha lógica, habría que realizar un migrado completo del sistema.
* A nivel de lógica, sería necesario que el administrador del sistema tuviese en su poder la cuenta de owner, lo cual no resulta muy lógico.

## Iteración 2

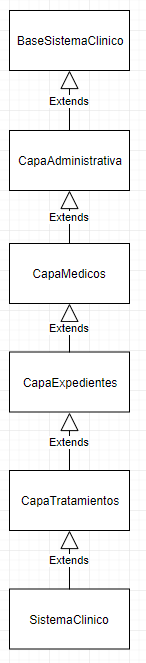
Encontradas las debilidades mencionadas, en esta segunda iteración se busca mejorarlas y se plantea lo siguiente:

* Separar la lógica de negocio de los datos, sacando estos últimos a un contrato separado. De este modo, en caso de tener que cambiar el contrato del sistema clínico por un bug, podría no ser necesario el migrado del contrato de los datos.
* Separar la lógica en distintos contratos abstractos con una jerarquía lógica. Aunque al final se use herencia para recopilar toda esa lógica en un único contrato, y en caso de tener un bug de uno de los abstractos habría que desplegar de nuevo el contrato “principal”, la cohesión a nivel de contrato sería más baja.
* Se mete un nuevo actor en el sistema, el **administrativo**, cuya labor es la de administrar el sistema a nivel funcional, encargándose este de dar de alta/baja los médicos, transferir expedientes de un medico a otro…

Para realizar esta aproximación se añaden los siguientes contratos:

* DatosSistemaClinico.sol: En él se almacenan todos los datos del sistema clínico que antes estaban en el mismo, como la lista de médicos, la de pacientes, la nueva de administrativos… Para independizar aún más los datos de la lógica, en vez de almacenar los expedientes como una representación del propio contrato, se opta por almacena la dirección.
* BaseSistemaClinico.sol: Contrato base en el que se incluyen las librerías útiles, el contrato de datos y cosas comunes a todos los contratos del sistema clínico.
* CapaAdministrativa.sol: En este contrato se implementa toda la lógica correspondiente a la gestión de los administrativos.
* CapaMedicos.sol: En este contrato se implementa toda la lógica correspondiente a la gestión de los médicos.
* CapaExpediente.sol: En este contrato se implementa toda la lógica correspondiente a la gestión de los expedientes.
* CapaTratamientos.sol: En este contrato se implementa toda la lógica correspondiente a la gestión de tratamientos.
* SistemaClinico.sol: Este contrato hereda todas las funcionalidades anteriores exponiéndolas en un único contrato de forma unificada.

La relación de herencia sería la siguiente:



Una vez montada toda la estructura, al ir a ejecutar los tests, se encuentra un problema bastante grande, al sacar los datos un contrato externo, el bytecode generado supera el tamaño de bloque, lo que imposibilita su despliegue. Esto debe ser porque el bytecode necesario para manejar una variable del propio contrato debe ser menor al generado para la interactuación con otro contrato.

## Iteración 3

Debido al problema anteriormente mencionado, se plantea una tercera iteración, en la cual, se tiene que resolver el problema del tamaño del contrato. Esto se aborda creando un facade por composición en vez de por herencia. Con esto conseguimos varias cosas:

* En primer lugar resolver el problema del tamaño del contrato, ya que al usar composición, el facade no contendrá la lógica de negocio de cada una de las capas, si no que la delegará al contrato correspondiente.
* Conseguimos una baja cohesión real, ya que ahora la lógica de cada capa si que va a ser completamente independiente del resto.
* Al inyectar al facade las diferentes capas, estamos disminuyendo la necesidad de migración del propio facade, ya que en caso de tener que cambiar el contrato de alguna de las capas, será tan sencillo como inyectar al facade el nuevo contrato (siempre y cuando siga cumpliendo la misma interfaz).

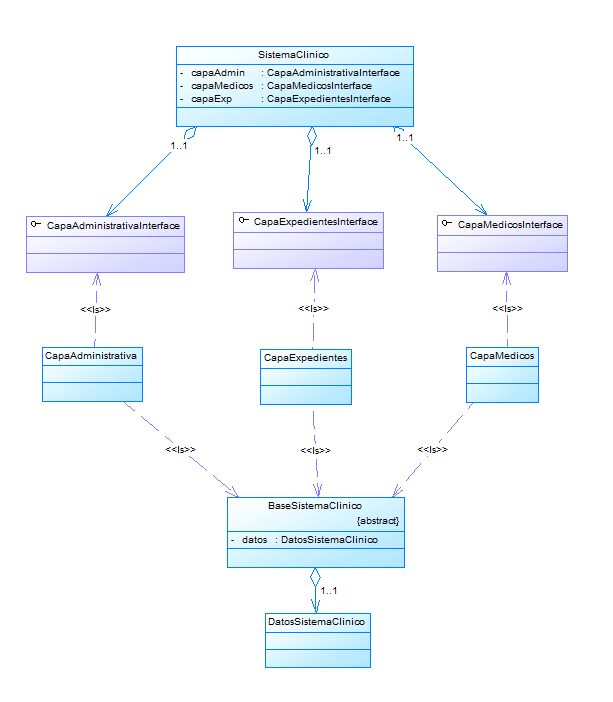
Con este modelo, el sistema clínico queda en una arquitectura de 3 niveles, donde el primero será un punto de entrada único al sistema clínico (facade), el cual orquestará a alto nivel a que contrato/s delegar las llamadas.

El segundo nivel lo formarán los distintos contratos que alojan la propia lógica de negocio (CapaAdministratica, CapaMedicos y CapaExpedientes). Dado al alto acoplamiento entre los anteriores contratos de expedientes y tramatamientos, se decide unificarlos en un único contrato, eliminado el contrato CapaTratamientos.

El tercer y último nivel lo compone el contrato que contiene todos los datos del sistema clínico.

Con esta arquitectura lo que estamos consiguiendo es una alta tolerancia a fallos, y una baja cohesión.

Sin entrar en mucho detalle de los métodos y propiedades de cada contrato, a continuación se expone un diagrama en el que se resumen las propiedades y relaciones de los contratos de los diferentes niveles para entender de un solo vistazo la arquitectura y la interactuación de sus diferentes componentes.



Además de todo esto, se implementa un proxy para obtener la última versión del sistema sanitario, de modo que si hubiese que migrarlo, para los usuarios del mismo sería completamente transparente.

## Posibles mejoras

Se podría hacer una mejora implementado el facade a través del patrón de diseño del contrato transparente y el ERC1538, lo cual te permitiría añadir y eliminar métodos de forma dinámica sin necesidad de tener que desplegar un nuevo facade.

## Conclusiones

Tras los problemas descritos en este documento, y otros muchos que se han obviado, se llega a la conclusión de que Ethereum como blockchain pública no es la plataforma ideal para el desarrollo de una aplicación de estas características.

Igual si que se podría usar para guardar parte de la información, como por ejemplo la auditoria de acceso a los expedientes, pero en ningún caso para albergar toda la información y lógica de negocio del sistema.

## Despliegue en rinkeby

SistemaClinico: 0x042cf1c635c9df34d2f9edbce90aaa727959e179

DatosSistemaClinico: 0xf1f024a98e21f873324c5ad264d1502ea4630051

CapaAdministrativa: 0x9899448aba6809e67efd929f2f6697ea63966267

CapaMedicos: 0xe5b79a83c3495d0cfbefea425cc023b3dcfd254a

CapaExpedientes: 0xba049c25d192ef29e69b01d9fdb2cc6de3fbe543