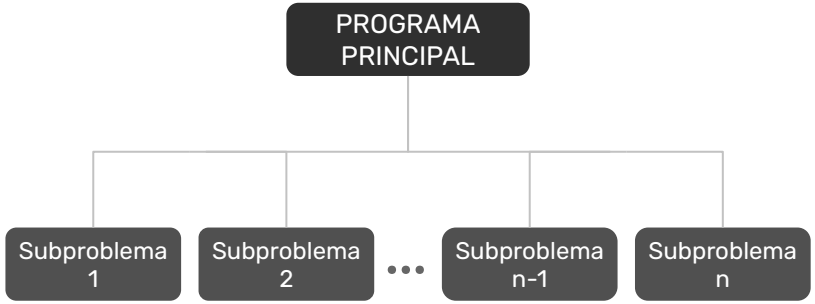
Desarrollo Blockchain Ethereum con Solidity

Módulo 4 - Buenas prácticas

Buenas prácticas

Modularización

- Mantener tus contratos **pequeños** y **fáciles** de entender.
- Separar las funcionalidades no relacionadas en otros contratos o en librerías.
- Limitar la cantidad de variables locales.
- Limitar la longitud de las funciones.
- Documentar las funciones para que otros puedan ver cuál era la intención del código y para ver si hace algo diferente de lo que pretendía.
- Aprovechar la herencia de contratos para reutilizar código.



Desarrollando “a prueba de fallos”

Aunque hacer que tu sistema sea completamente descentralizado eliminará cualquier intermediario, puede que sea una buena idea, especialmente para nuevo código, incluir un sistema a prueba de fallos.

Puedes agregar una función a tu contrato que realice algunas comprobaciones internas como *“¿Se ha filtrado Ether?”, “¿La sumatoria de los tokens es igual al account balance?”*, etc.

Recordar que no se puede usar mucho gas para eso, así que ayuda mediante computaciones **off-chain*** podrían ser necesarias.

Si los chequeos fallan, el contrato automáticamente cambia a modo a prueba de fallos, donde, por ejemplo, se desactivan muchas funciones, da el control a una entidad tercera de confianza o se convierte en un contrato del tipo *“devolveme mi dinero”.*

No confiar en los datos de bloque

Hay que tener en cuenta que los mineros escriben la información del bloque.

Si nuestro contrato depende de datos de bloque como el timestamp, puede que éste sea modificado y afecte el funcionamiento.

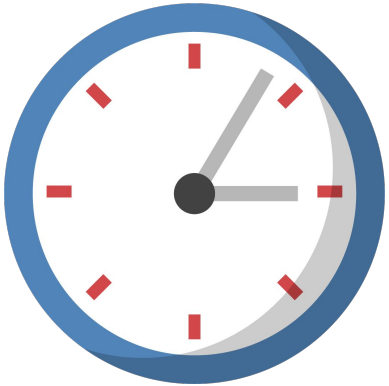
También es posible este escenario para la generación de números aleatorios.

Speed bump pattern

Algunas operaciones críticas pueden verse afectadas por un número alto de concurrencia de usuarios.

Un patrón que se utiliza en estos casos es el de aplicar una demora en la ejecución de la operación sensible.

Por ejemplo, un retiro de fondos se puede realizar a los 5 días de haberse pedido.



Control de acceso

Es conveniente que cada función tenga un control de acceso.

Podemos definirlo vía modifiers o bien usar alguna librería que nos brinde esa funcionalidad.

Guía de estilos de codificación

Plantear una guía de estilos de codificación pretende proporcionar convenciones de codificación para escribir código con Solidity.

Esta guía debe ser entendida como un documento en evolución que cambiará con el tiempo según aparecen nuevas convenciones útiles y antiguas convenciones se vuelven obsoletas.

```
/**
* Code Readability
*/
if (readable()) {
    be_happy();
} else {
    refactor();
}
```

Entre las principales encontraremos:

- Diseño del código.
- Codificación de archivos de origen.
- Orden de funciones.
- Espacios en blanco en expresiones.
- Estructuras de control.
- Declaración de funciones.
- Recomendaciones generales.

Guía de estilos de codificación

Diseño del código

- **Sangría:**
  Utilice 4 espacios por nivel de sangría.
- **Tabulador o espacios:**
  Los espacios son el método de indentación preferido.
  Debe evitarse la mezcla del tabulador y los espacios.
- **Líneas en blanco:**
  Envuelva las declaraciones de nivel superior en el código de Solidity con dos líneas en blanco.

```
contract A {
    ...
}


contract B {
    ...
}


contract C {
    ...
}
```

Guía de estilos de codificación

Codificación de archivos de origen

- Se prefiere la codificación del texto en UTF-8 o ASCII.
- Las declaraciones de importación siempre deben colocarse en la parte superior del archivo.
- No se debe realizar imports entre funciones.

```
import "owned";

contract A {
    ...
}

contract B is owned {
    ...
}
```

Guía de estilos de codificación

Orden de funciones

El orden ayuda a que los lectores puedan identificar las funciones que pueden invocar y encontrar las definiciones de constructor y de retorno más fácilmente.

Las funciones deben agruparse de acuerdo con su visibilidad y ser ordenadas:

- constructor.
- función fallback (de existir).
- external.
- public.
- internal.
- private.

```
contract A {
    function A() {
        ...
    }

    function() {
        ...
    }

    // Funciones external
    // ...

    // Funciones external que son constantes
    // ...

    // Funciones public
    // ...

    // Funciones internal
    // ...

    // Funciones private
    // ...
}
```

Guía de estilos de codificación

Espacios en blanco en expresiones

Evitar los espacios en blanco superfluos en las siguientes situaciones:

- Inmediatamente entre paréntesis.
- Llaves o corchetes (con la excepción de declaraciones de una función en una sola línea).

```
spam(ham[1], Coin({name: "ham"}));
```

```
function singleLine() { spam(); }
```

```
function spam(uint i, Coin coin);
```

Guía de estilos de codificación

Estructuras de control

Las llaves que denotan el cuerpo de un contrato, biblioteca, funciones y estructuras deberán:

- Abrir en la misma línea que la declaración.
- Cerrar en su propia línea en el mismo nivel de sangría que la declaración.
- La llave de apertura debe ser procedida por un solo espacio.

```
contract Coin {
    struct Bank {
        address owner;
        uint balance;
    }
}
```

```
if (...) {
    ...
}

for (...) {
    ...
}
```

Guía de estilos de codificación

Declaración de funciones

- Para declaraciones de función cortas, se recomienda dejar el corchete de apertura del cuerpo de la función en la misma línea que la declaración de la función.
- El corchete de cierre debe estar al mismo nivel de sangría que la declaración de la función.
- El corchete de apertura debe estar precedido por un solo espacio.

```
function increment(uint x) returns (uint) {
    return x + 1;
}

function increment(uint x) public onlyowner returns (uint) {
    return x + 1;
}
```

Guía de estilos de codificación

Recomendaciones generales

- Los strings deben de citarse con comillas dobles en lugar de comillas simples.

```
str = "foo";
str = "Hamlet says, 'To be or not to be...'";
```

- Se envuelve los operadores con un solo espacio de cada lado.

```
x = 3;
x = 100 / 10;
x += 3 + 4;
x |= y && z;
```

Guía de estilos de codificación

Recomendaciones generales

- Para los operadores con una prioridad mayor que otros, se pueden omitir los espacios de cada lado del operador para marcar la precedencia.

```
x = 2**3 + 5;
x = 2*y + 3*z;
x = (a+b) * (a-b);
```

¡Muchas gracias!

¡Sigamos trabajando!