Evaluación 3

Principios Solid

NOMBRE: Matias Valdes Reyes, Javier Bobadilla.
CARRERA: Ing. informática
ASIGNATURA: ingeniería de Software
PROFESOR: Pablo Cerda Wehinger.
FECHA: 13-12-23

Normativas de buenas prácticas:

Principio Single Responsability:

Esta funcionalidad cumplía con 5 funciones:

1. Realizar un ciclo For por cada fabrica validando la entrada de datos con números positivos.
2. Validar la entrada de datos, en donde se exige que el numero sea positivo.
3. Calcular la producción total.
4. Agregar las fabricas a una lista para luego utilizar los valores de cada una.
5. Cambiar del modal Fabricas al modal Bodegas.

```
okFab.addEventListener('click',(e)=>{
    e.preventDefault();
    produccionTotal=0;
    for( var i=1; i<Number(cantFabricas)+1;i++){
        if(Number(document.getElementById('fab'+i).value)<1){
            alert("Favor sólo ingresar números positivos")
            return;
        }
        produccionTotal=(Number(document.getElementById('fab'+i).value))+produccionTotal;
        fabricas.push(new fabrica(i,Number(document.getElementById('fab'+i).value)))
    }
    containerModalRootFab.style.display='none'
    cantFab.style.display='none'
    containerModalRootBod.style.display='block';
    console.log(produccionTotal)

})
```

Luego de aplicar el principio se realizaron clases con métodos específicos para cada responsabilidad.

1. Class ControllerFabricaAndProduccion se encarga de controlar e iterar las demás entidades.
2. Class ValidateMayor se encarga de validar los datos ingresados, en ese caso que la producción por cada fabrica sea positiva.
3. Class Producciónтotal se encarga de calcular la producción total sumando la producción de todas las fábricas.
4. Class FabricasTotales se encarga de agregar a la lista todas las fábricas ingresadas.
5. Class ChangeModalFabricasToBodegas se encarga de realizar el cambio de modal para así solicitar la información de las bodegas.

```
// CLASS FABRICAS

class ControllerFabricaAndProduccion{
    controllerFabricaAndProduccion(){
        p.produccion=0;
        f.fabricas=[]
        for( var i=1; i<Number(cantFabricas)+1;i++){
            if(validation.validateNumberPositive(Number(document.getElementById('fab'+i).value))===null)return
            p.calculateProduccionTotal(Number(document.getElementById('fab'+i).value));
            f.addFabricaToList(new fabrica(i,Number(document.getElementById('fab'+i).value)))
        }
        changeModalFab.changeModalFabricasToBodegas()
    }
}

class ValidateMayor{
    number=1
    validateNumberPositive(numberCompare){
        if(this.number>numberCompare){
            alert("Favor sólo ingresar números positivos")
            return null;
        }
    }
}

class ProduccionTotal{
    produccion=0;
    calculateProduccionTotal(produccionPlus){
        this.produccion=this.produccion+produccionPlus;
    }
}

class FabricasTotales {
    fabricas=[]
    addFabricaToList(fabrica){
        this.fabricas.push(fabrica)
    }
}

class ChangeModalFabricasToBodegas{
    changeModalFabricasToBodegas(){
        containerModalRootFab.style.display='none'
        cantFab.style.display='none'
        containerModalRootBod.style.display='block'
    }
}
```

Principio Open/Closed:

Esta función queda abierta al cambio debido a que si incorpora un nuevo botón se debería modificar, esto rompe el principio.

```
function getInput(button){
    if (button.value=='btnBod'){
        containerBod.innerHTML=""
        cantFab.style.display="none"
        cantBod.style.display="block"
        containerModalRootBod.style.display="none";
        cantBodegas=button.value;
        Inputs(cantBodegas,'bod','Bodega',containerBod)
    }else if(button.value=='btnFab'){
        containerFab.innerHTML=""
        cantFab.style.display="block"
        containerModalRootFab.style.display="none";
        cantFabricas=button.value;
        Inputs(cantFabricas,'fab','Fábrica',containerFab)
    }
}
```

En este ejemplo se logra apreciar que se agrego una nueva funcionalidad sin modificaciones.

```
var cantFabricas=0;

function getInputsFab(button){
    containerFab.innerHTML=""
    cantFab.style.display="block"
    containerModalRootFab.style.display="none";
    cantFabricas=button.value;
    Inputs(cantFabricas,'fab','Fábrica',containerFab)
}

var cantBodegas=0;

function getInputsBod(button){
    containerBod.innerHTML=""
    cantFab.style.display="none"
    cantBod.style.display="block"
    containerModalRootBod.style.display="none";
    cantBodegas=button.value;
    Inputs(cantBodegas,'bod','Bodega',containerBod)
}
```

Principio Dependency Inversion:

Esta clase depende de ControllerBodegaAndAlmacenamiento, esto viola el principio.

```
class Controller{
    constructor(){
        this.control=new ControllerBodegaAndAlmacenamiento;
    }
}
```

Esto fue reemplazado por el siguiente ejemplo, donde el constructor solo espera el argumento que esto definirá que Controller asumirá.

```
class Controller{
    constructor(controller){
        this.controller=controller;
    }
}


okFab.addEventListener('click',(e)=>{
    e.preventDefault();
    const c=new Controller(controllerFP);
    c.controller.controllerFabricaAndProduccion();

})

okBod.addEventListener('click',(e)=>{
    e.preventDefault();
    const c= new Controller(controllerBA)
    c.controller.ControllerBodegaAndAlmacenamiento()
})
```

Conclusión:

Los principios SOLID dan un orden y resuelven problemáticas recurrentes en el desarrollo, no obstante, el lenguaje JavaScript no da ejemplos muy claros de su uso. Esto es debido a que su uso no es orientado a objetos, y es débilmente tipado y no se pueden crear interfaces para esto se utiliza más el lenguaje TypeScript.