Ejercicio 1

El código viola el Principio de Responsabilidad Única (SRP), tras revisar exhaustivamente el código se ha llegado a la conclusión de que este estaba realizando múltiples tareas en un mismo método, cosa que hacía difícil de leer el código y complicaba el entendimiento de cómo funcionaba. De cara a la refactorización se ha separado en varios métodos las diferentes funciones y clases (EmplyeeFormater) que la clase debe cumplir para poder alcanzar el resultado deseado.

Ejercicio 2

El código viola el Principio de Responsabilidad Única (SRP), una vez más al revisar el código se ha determinado que la clase *CarManager()* estaba realizando múltiples tareas, cosa que no debería de ser ya que complica el entendimiento al leer el código. Para la refactorización se ha separado la creación de la "base de datos" de coches y el formateo para mostrarlo en dos clases distintas.

- **CarDatabase:** Una clase con un array de coches como parámetro, con u constructor para instanciar la DB.
- **CarManager:** Una clase con métodos para mostrar la información de un coche por ID o todo el listado de coches.
- **CarFormatter:** Una clase que solo cuenta con dos métodos. *formatCars($car)* con el cual se le pasa por parámetro un array de objetos "Car" y lo formatee para posteriormente mostrarlo por pantalla. Y el método *getBestCar($cars),* el cual selecciona el mejor coche del array de objetos "Car" en base a su posición alfabética (A como menor, Z como mayor).

Ejercicio 3

El código viola el Principio de Abierto/Cerrado (OCP) ya que la existencia de la clase *"CalculateArea()"* implica que a la hora de que el código se tenga que extender, se deberá modificar esta clase para añadir el método de cálculo de nuevas formas. En la refactorización, se ha priorizado mover los métodos de cálculo de área a su correspondiente forma y creado un método vacío en la clase abstracta, para que a medida que se vayan añadiendo nuevas formas, estas cumplan con el mismo estándar.

Ejercicio 4

El código viola el Principio de Sustitución de Liskov (LSP), ya que la clase cuadrado hereda de la clase rectángulo, forzando a que el ancho y el alto sean iguales, cuando deberían actuar como en Rectángulo al tratarse de una herencia. Esto se soluciona en la refactorización al hacer a Cuadrado y Rectángulo clases independientes con sus propiedades y métodos únicos.

Ejercicio 5

El código no viola ninguno de los Principios SOLID, sin embargo, la clase "*Pool()"* se encuentra bastante cargada de responsabilidades por lo que de seguir creciendo, podría llegar a violar el Principio de Responsabilidad Única (SRP). Es por ello por lo que en la refactorización lo que se ha hecho ha sido separar la gestión de patos con la piscina en sí, de manera que, en caso de ampliarse el programa, este sea más escalable a futuro.

Ejercicio 6

El código viola el Principio de Sustitución de Liskov (LSP), ya que la clase "*Robot()"* no cumple con lo que se espera que haga la interfaz Worker ya que un Robot no puede comer. Para la refactorización se ha optado por crear dos Interfaces (Worker y Eater), he implementarlas ambas a Human mientras que Robot solo posee Worker.

Ejercicio 7

El código no viola ninguno de los Principios SOLID, sin embargo, podría llegar a violar el Principio de Responsabilidad Única (SRP) al cargar con la responsabilidad de controlar los Callbacks de las puertas. Ya que todas las puertas no cuentan con las mismas Callbacks, se ha separado lo que es la interfaz de la puerta y las llamadas en dos interfaces (TimerClient y SensorClient) diferentes para la refactorización, para que en caso de añadir una puerta sin Callbacks, la clase no deba tener métodos vacíos.

Ejercicio 8

El código viola el Principio de Responsabilidad Única (SRP), ya que en la clase *"Lamp()"* está establecida la propiedad color, con sus respectivos getters y setters, los cuales no se usan nunca en ninguna parte del programa. De cara a la refactorización lo que se ha hecho ha sido borrarlo.

Ejercicio 9

El código viola el Principio de Inversión de Dependencia ya que la clase *"EmplyeeDetails()"* está dependiendo directamente de la clase *"SalaryCalculator()",* no solo eso, sino que también las clases carecen de un constructor, las propiedades en *"EmplyeeDetails()"* son públicos y se hace llamado a un método para llamar a otro método. En la refactorización se ha solventado todo esto agrupándolo en una misma clase, con un constructor, getters y la función CalculateSalary() originalmente ubicada en *"SalaryCalculator()".*