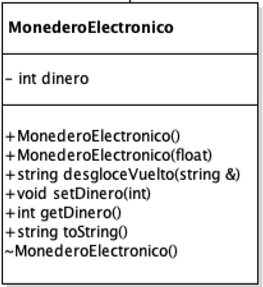
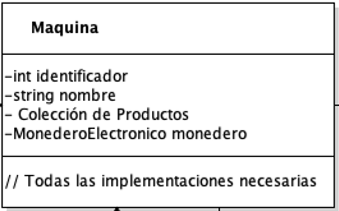
**Principios SOLID utilizados en el Proyecto 1**

**Responsabilidad única:**

Este principio indica que una clase o método debería de cumplir una única función y no tener responsabilidades mixtas. Esto debe respetarse ya que las clases podrían acabar teniendo muchas responsabilidades a la vez. Un ejemplo claro de esto en el proyecto sería el caso de la clase maquina y la clase monedero electrónico.



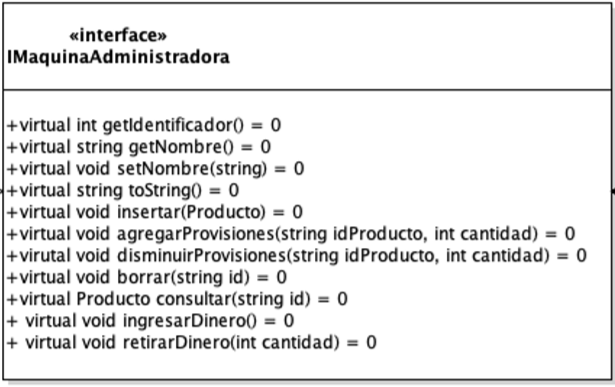
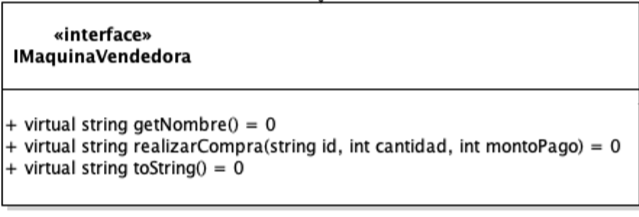
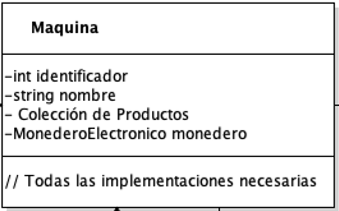
En este caso la clase maquina se encargará de administrar el grupo de objetos “Producto”, podrá manejar una lista de los mismos y la logística de la compra sin embargo parte de lo que el problema requiere es que la maquina tenga la capacidad de almacenar dinero y que cuando un usuario realice una compra esta máquina tenga la capacidad de poder dar un vuelto con la menor cantidad de monedas o billetes.

Dichas funciones sin embargo pertenecen a una lógica completamente distinta a la que se supone que la clase maquina debería de tener. Por esta razón se creó una clase nueva llamada monedero, cuyas funciones serán almacenar dinero y dar vuelto. Dicha clase tendrá una relación de composición con la clase máquina, de esta manera maquina podrá acceder a las funciones almacenar dinero y vuelto manteniendo las responsabilidades separadas.

Por otro lado, cada una de las otras clases poseen métodos y atributos que respetan el principio de responsabilidad única y procuran no ocuparse de responsabilidades que no le competen.

**Abierto-Cerrado:**

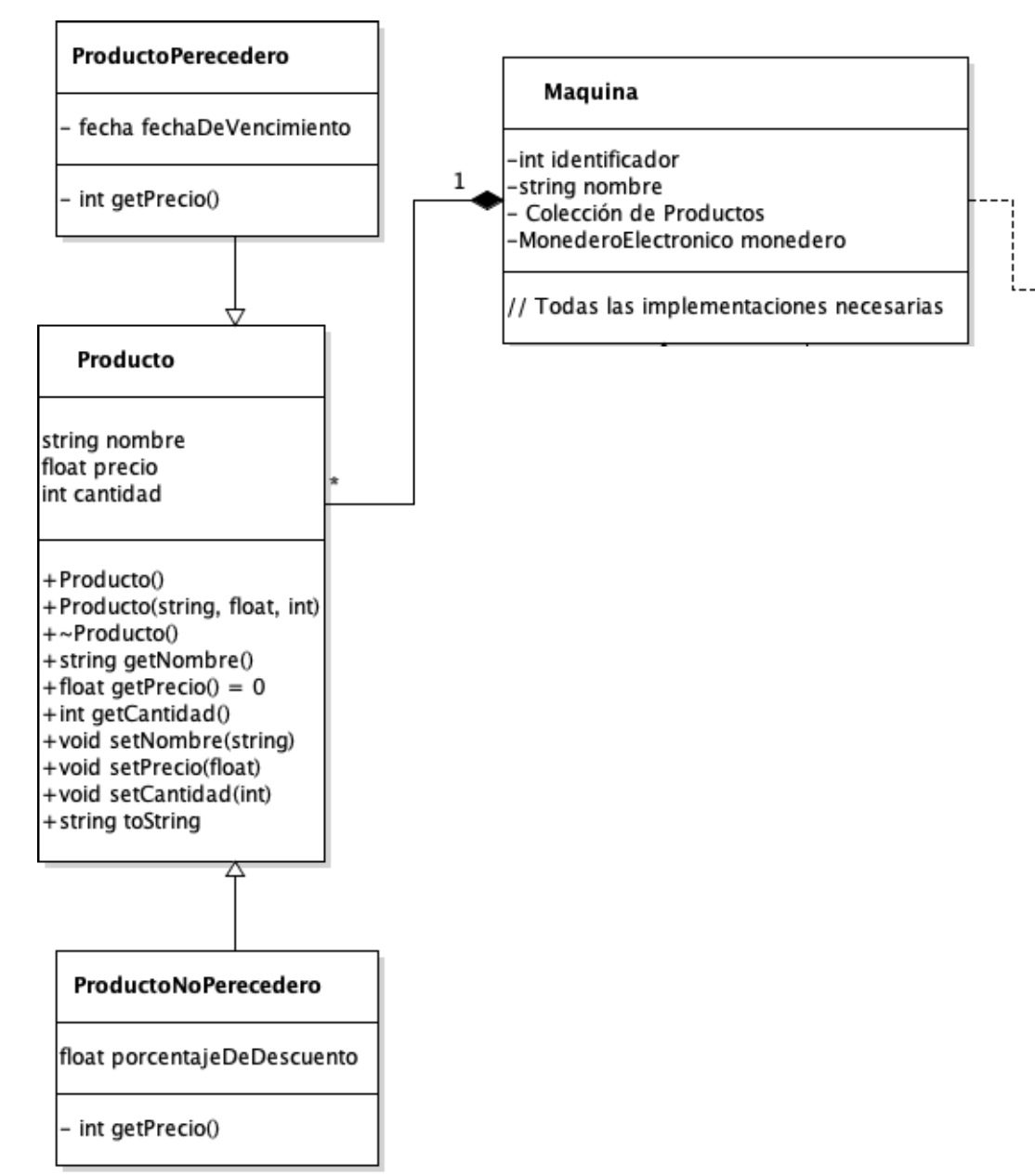
Este principio indica que las clases deberían de estar abiertas a la extensión y cerradas a la modificación. En el caso de este proyecto esto se puede ver reflejado en la clase máquina, dicha clase se puede extender mediante las interfaces que implementa. En este caso, la clase maquina podría tener funciones de maquina administradora únicamente, sin embargo, debido a la extensión, la misma maquina puede convertirse en una máquina vendedora, según se requiera. Por otro lado, cada una de las interfaces le permiten adquirir nuevas funcionalidades si así se requiriera. Por lo tanto, la extensión de funcionalidades es posible sin necesidad de modificar el código original de cada módulo.

**Inversión de dependencias:**

Dicho principio indica que las clases de alto nivel no deberían depender de las clases de bajo nivel, ambas deberían de trabajar con abstracciones. Un claro ejemplo de esto en el proyecto se refleja en la dependencia que tiene la clase Maquina con la clase Producto.

ProductoPerecedero y ProductoNoPerecedero, son clases concretas, mientras que la clase Producto es abstracta. De manera que, es realmente ventajoso que Maquina depende de la abstracción de Producto y así pueda instanciar tanto ProductoPerecedero como ProductoNoPerecedero y nunca darse cuenta qué tipo de productos contiene en su lista. Asimismo, la clase Maquina también tiene una clase lista que bien podría estar instanciando una lista tipo ArrayList o una lista tipo LinkedList y nunca darse cuenta gracias al poder de ocultamiento que este principio ofrece.



**Segregación de Interfaces:**

Este principio nos obliga a no depender de métodos que no se ocupan. En este proyecto se utiliza la segregación de interfaces dado que cada uno de los menús (Administrador y Cobro) dependen respectivamente de las interfaces IMaquinaAdministradora y IMaquinaVendedora, de esta manera, el MenuAdministrativo no ve métodos que se implementan en Maquina que no le competen. Análogamente, el MenuCobro, no ve métodos que no le interesa conocer.

