**Actividad 2 - Implementación de principios SOLID y Grasp**

En la programación orientada a objetos existen cinco principios básicos (SOLID) que, bien aplicados, marcan la diferencia entre un buen y un mal diseño. La diferencia entre una aplicación fácil de mantener y una que no. La diferencia entre un buen desarrollador y uno que no lo es. Estos principios son:

* Single Responsability Principle (Principio de responsabilidad única)
* Open/Closed Principle
* Liskov Substitution Principle
* Interface Segregation Principle
* Dependency Inversion Principle

**Single Responsability Principle (Principio de responsabilidad única)**

Como su propio nombre indica, este principio habla de que una clase o módulo debe tener una única responsabilidad. Cumpliendo este principio se aseguras que la clase o módulo tiene una cohesión alta, lo que significa que su clase no hace más de lo que debería hacer. En definitiva, una única razón para cambiar. Si, por lo contrario, construye una clase con más de una responsabilidad, lo que está haciendo es acoplar dichas responsabilidades, y esto conduce a un diseño frágil y difícil de mantener, con todo lo que ello conlleva.

Violando este principio

|  |  |
| --- | --- |
|  |  |
| Respetando el Principio de Responsabilidad Única dividiendo las clases ya que es una herramienta indispensable para proteger el código frente a cambios, ya que implica que sólo debería haber un motivo por el que modificar una clase. |  |

**Principio de Abierto/Cerrado (Open/Closed Principle),** que dice que el código debería estar abierto para extenderlo y para añadirle nuevas funcionalidades, pero en cambio debería estar cerrado a modificaciones, salvo las modificaciones que se deban realizar si se encuentra algún error.

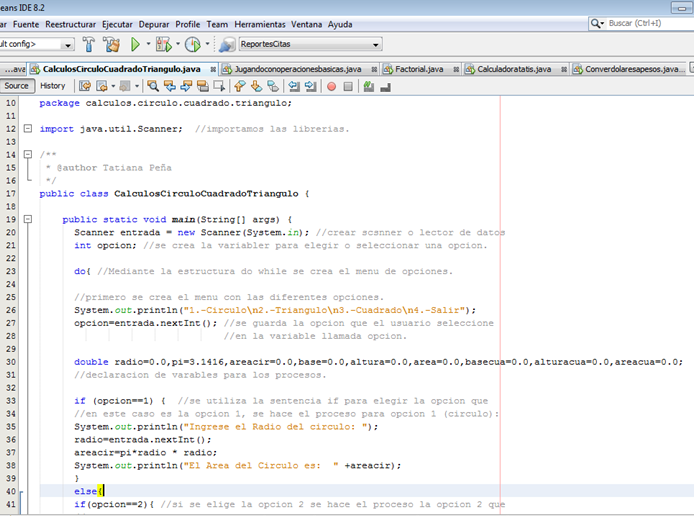
**Abiertos para la extensión**

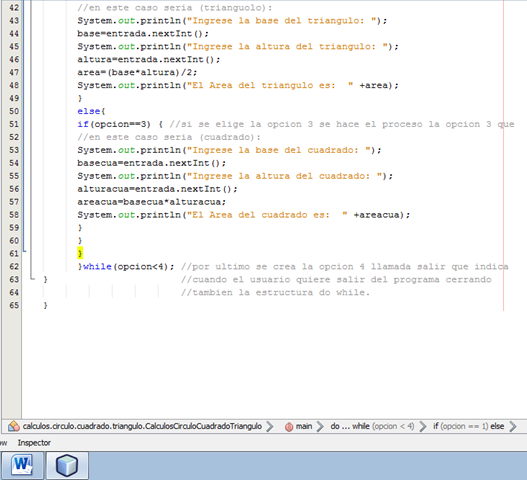
Esto significa que el comportamiento del módulo puede ser extendido. Cuando los requerimientos de la aplicación cambian, debemos ser capaces de extender el módulo con estos nuevos comportamientos que satisfagan esos cambios. En otras palabras, debemos ser capaces de cambiar lo que el módulo hace.

**Cerrado para la modificación**

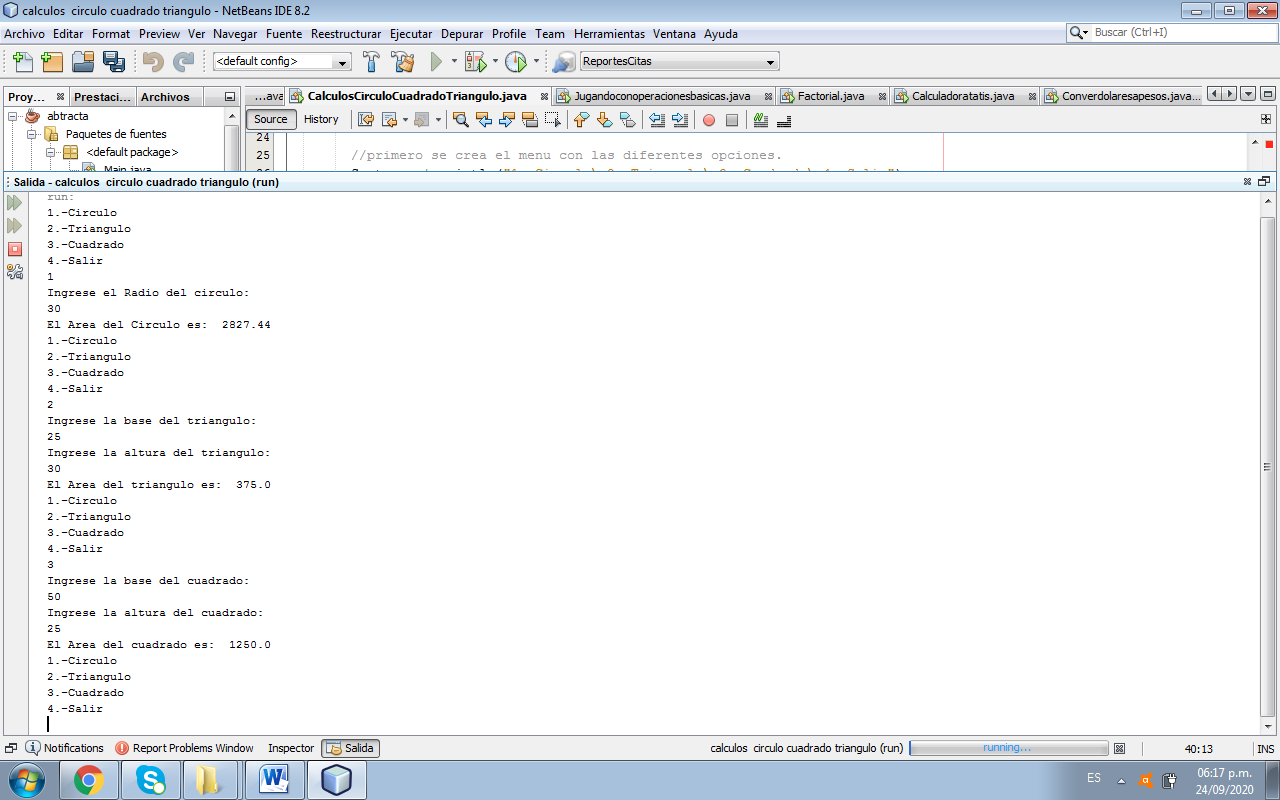
Esto significa que extender el comportamiento de un módulo no debería tener como resultado cambiar el código fuente, es decir, el código original debe permanecer sin cambios.

Ejemplo de violación

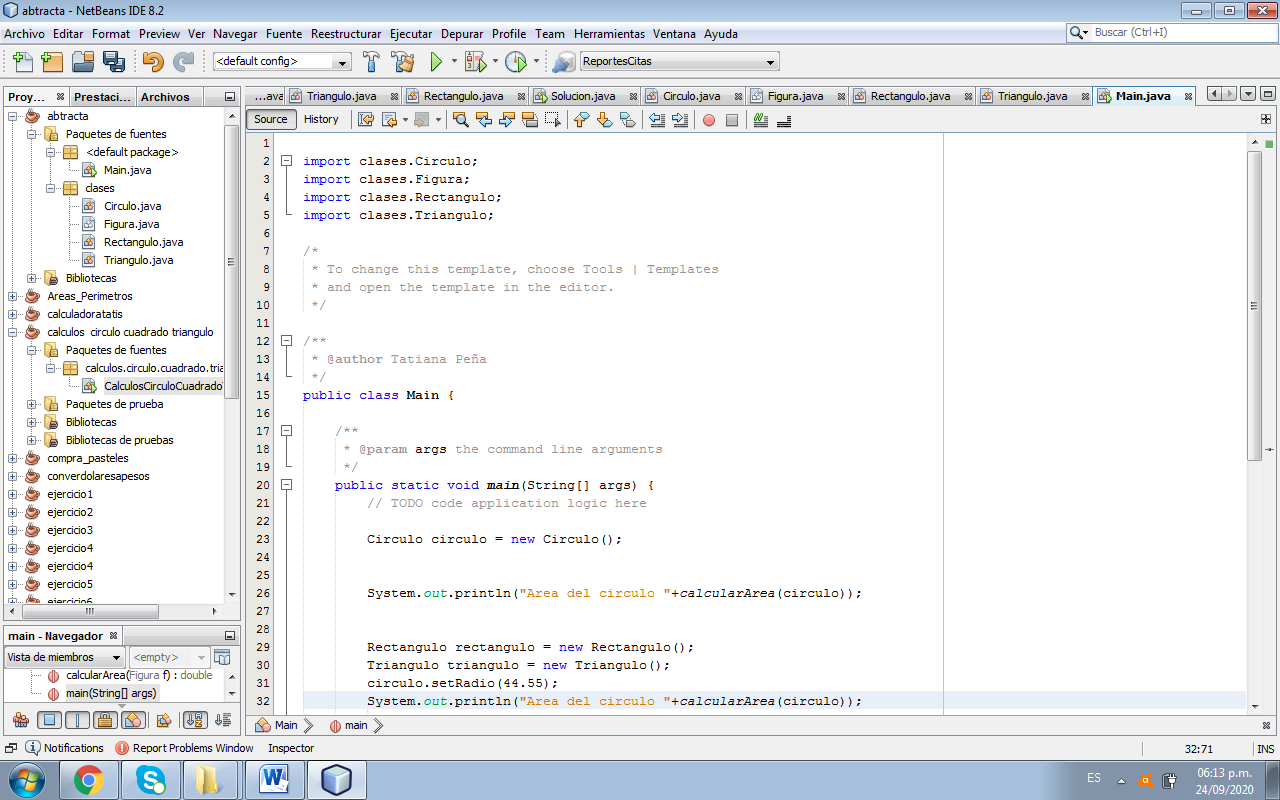


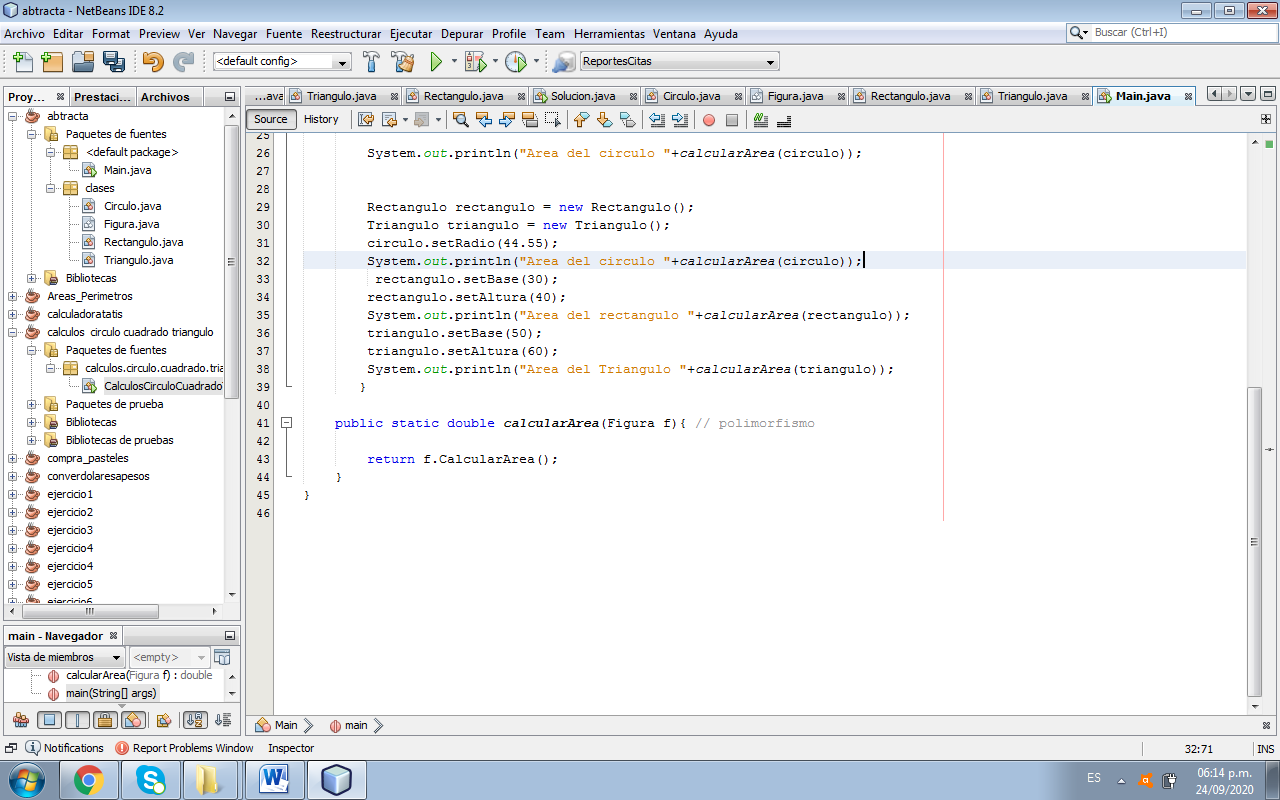


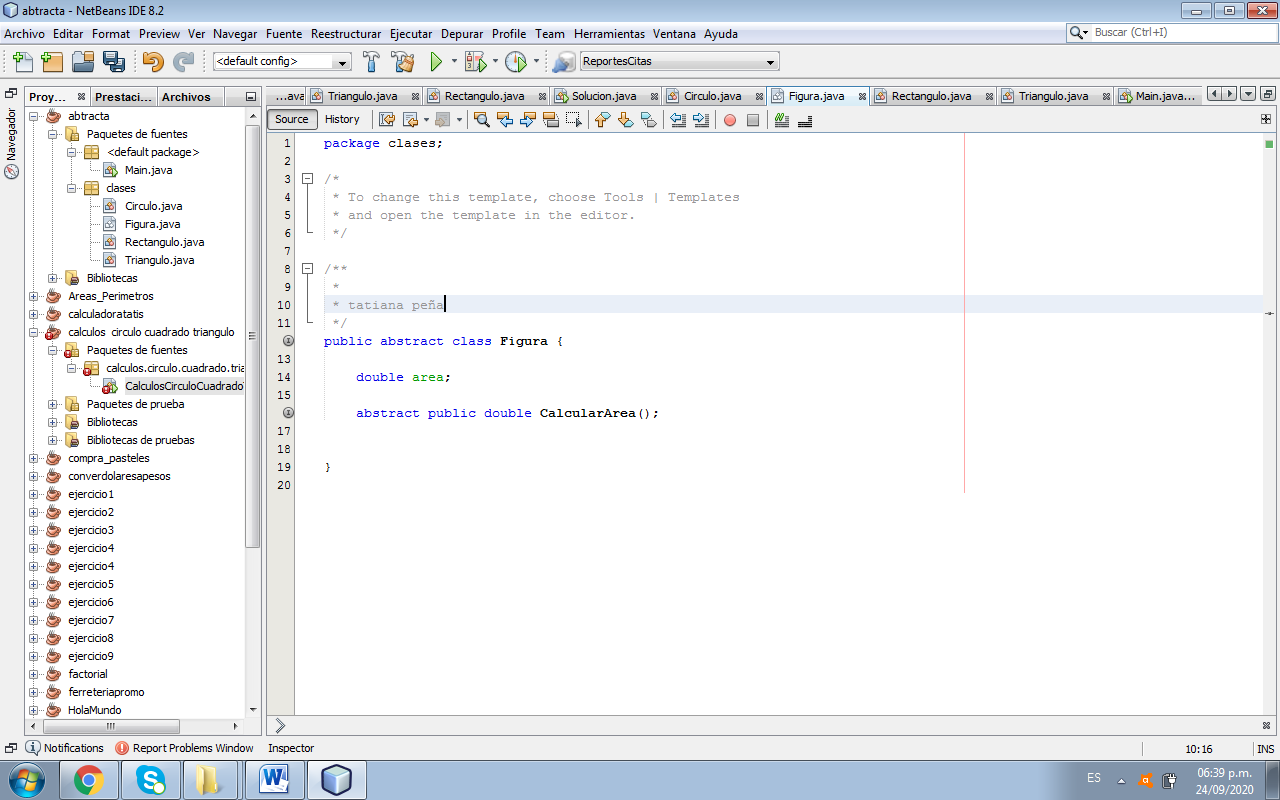
Al compilar

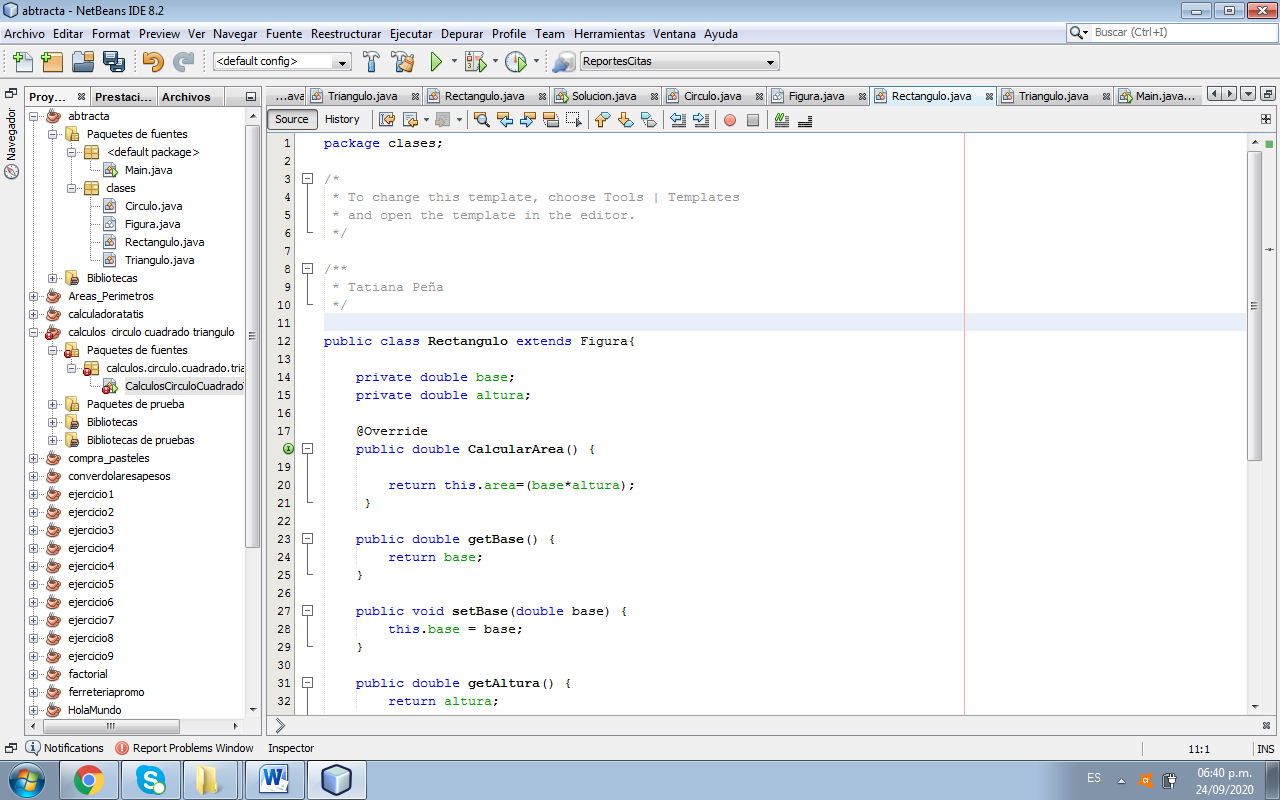


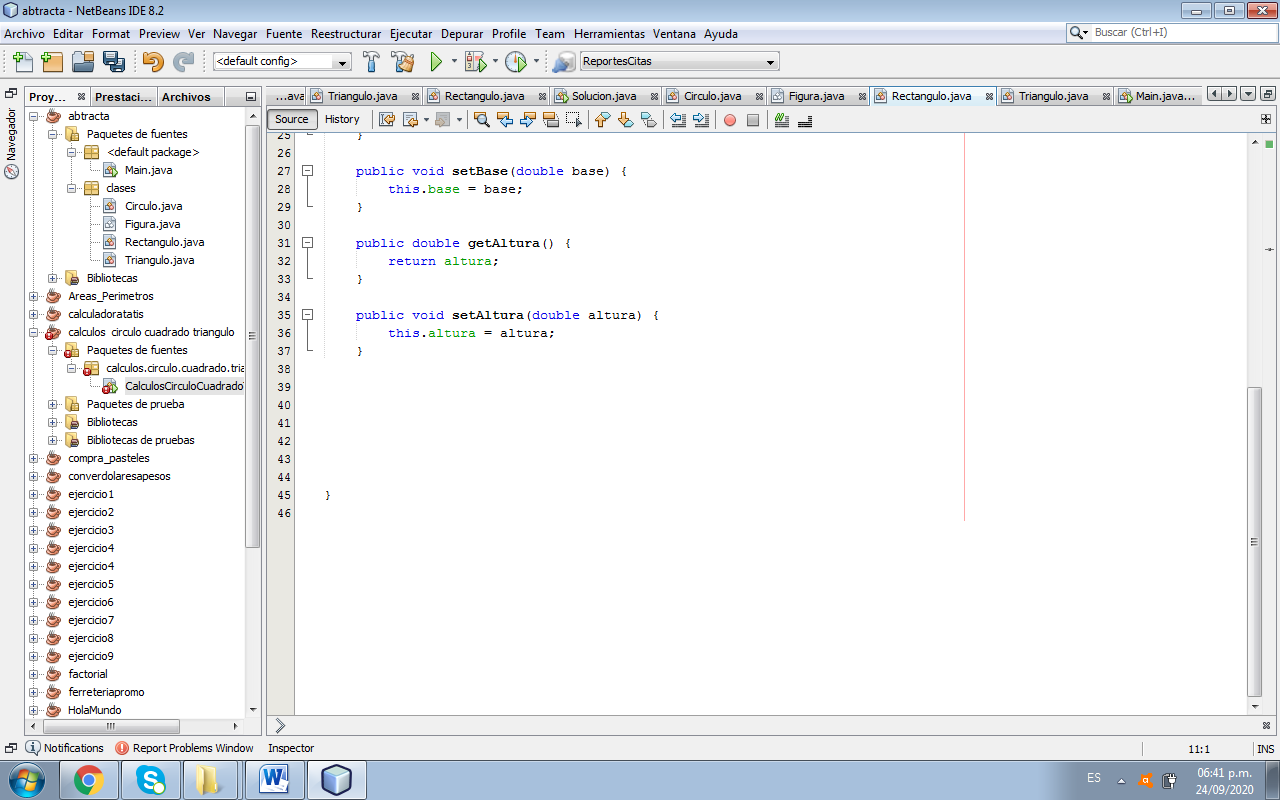
Al agregar la abstracción, la clase Figura cumple con el principio abierto-cerrado. Está cerrada para su modificación ya que tenemos la abstracción fija figura. Además, se encuentra abierta para su extensión ya que al momento de querer agregar una nueva figura podemos derivar la clase correspondiente a la figura, sin la necesidad de cambiar el código original de calcular áreas como se ve en el ejemplo.

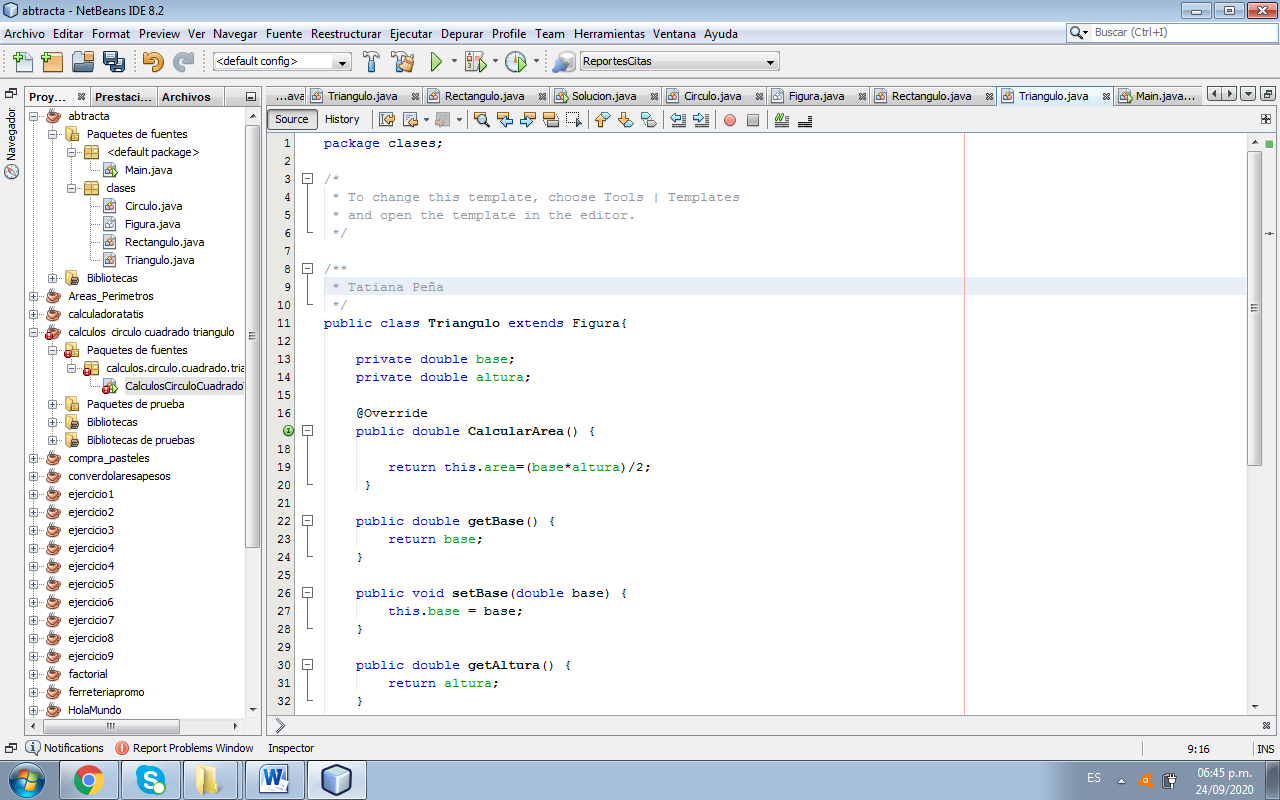


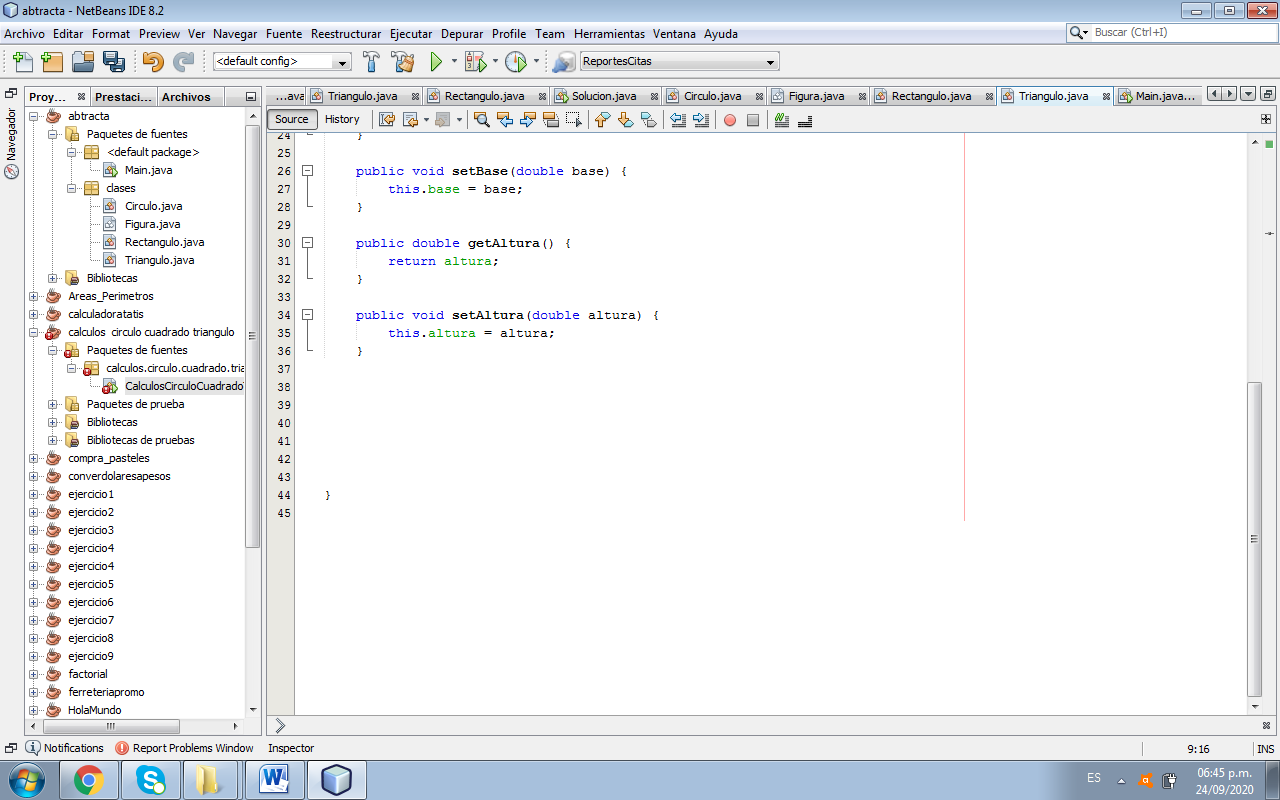




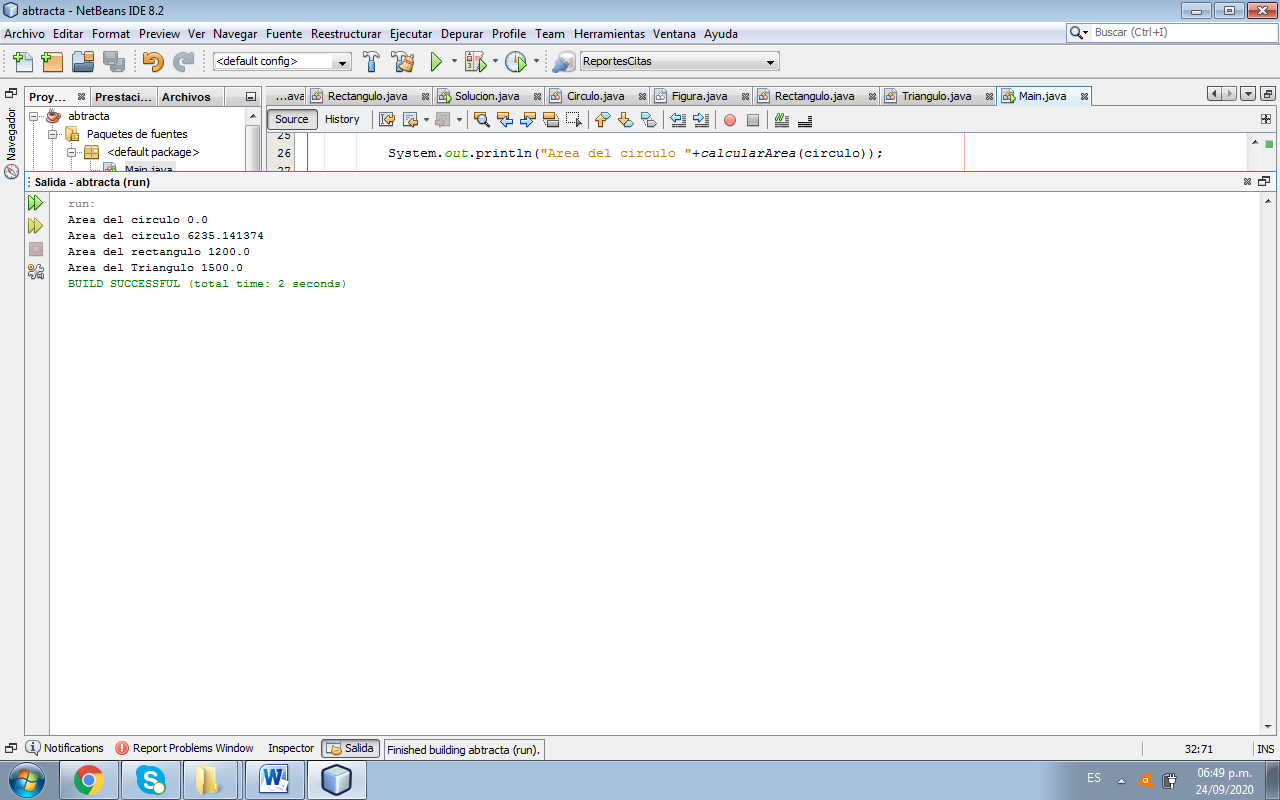








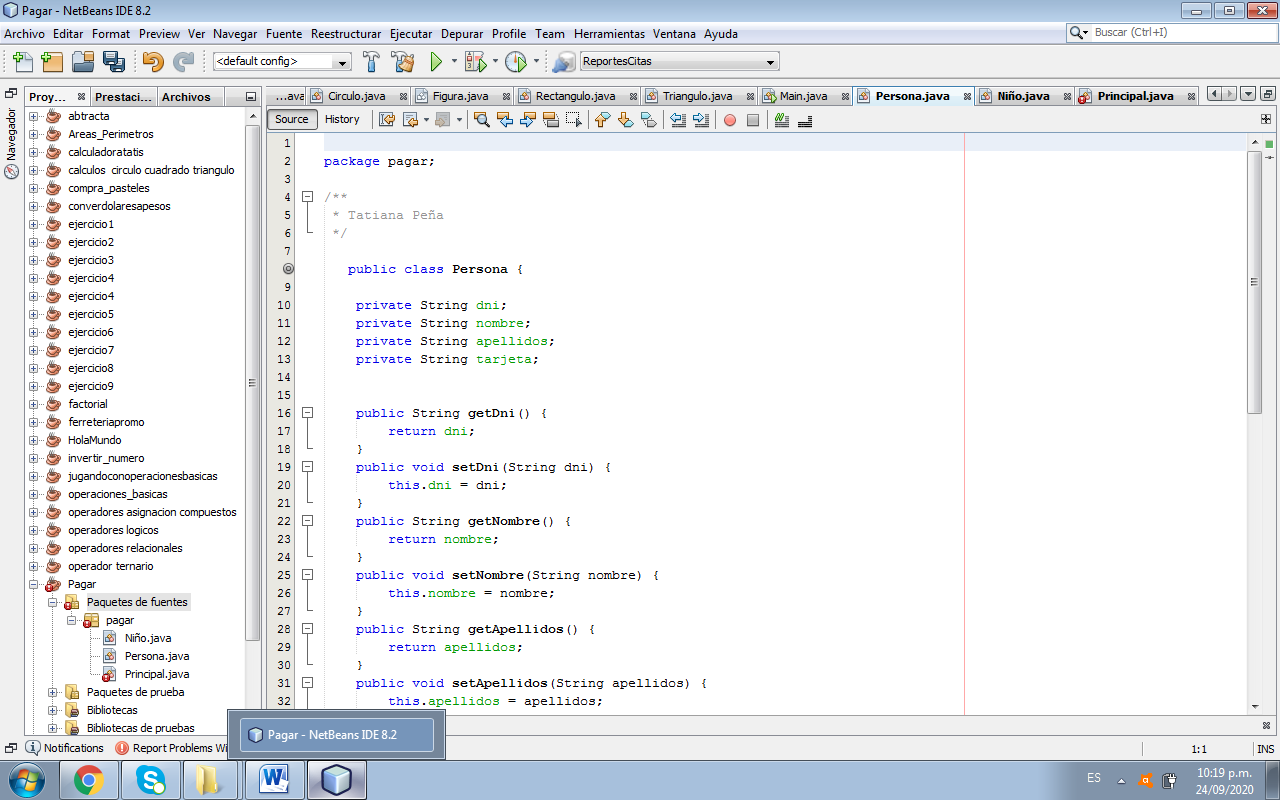
Compilando

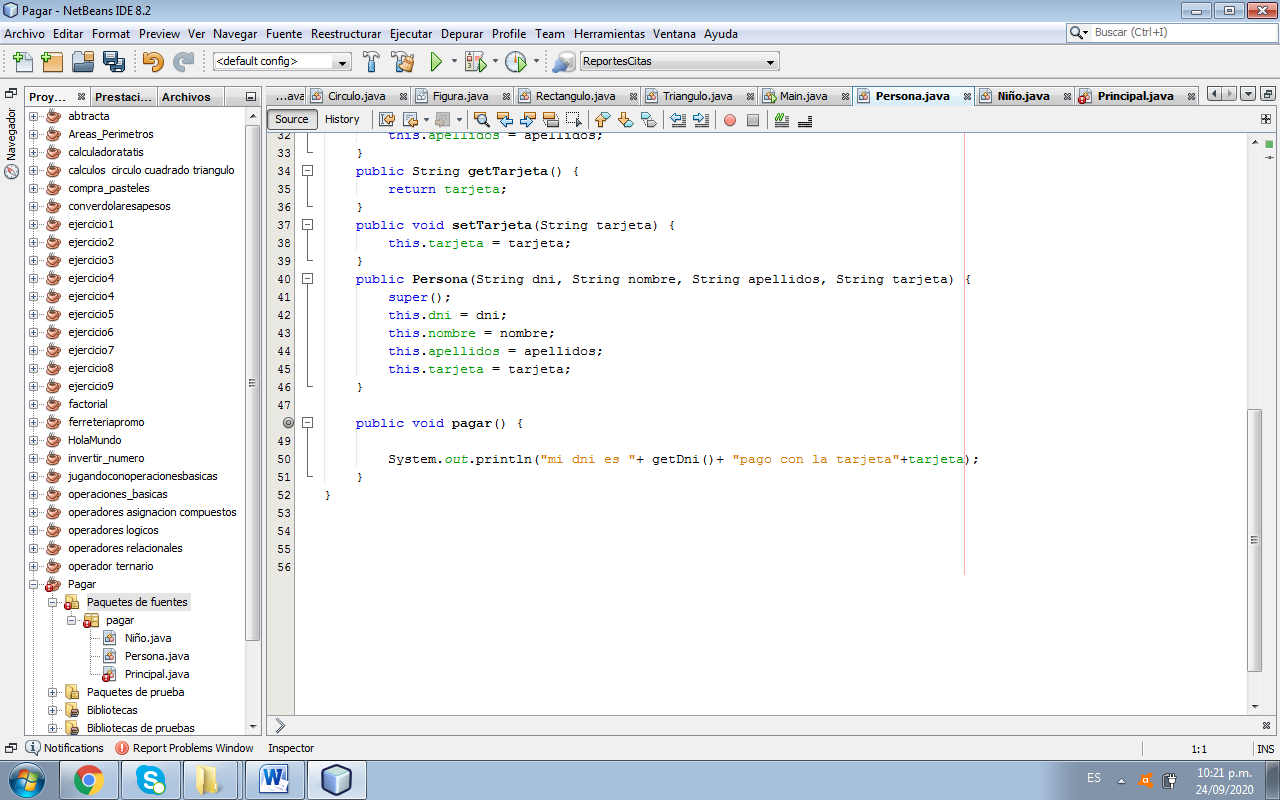


**Principio de Sustitución de Liskov (Liskov Substitution Principle),** que en resumen lo que dice es que una clase derivada debe poder ser sustituida por su clase base. Es una colección de pautas para crear jerarquías de herencia en las cuales un cliente podrá utilizar de manera confiable cualquier clase o subclase (clase hija) sin comprometer el comportamiento esperado.

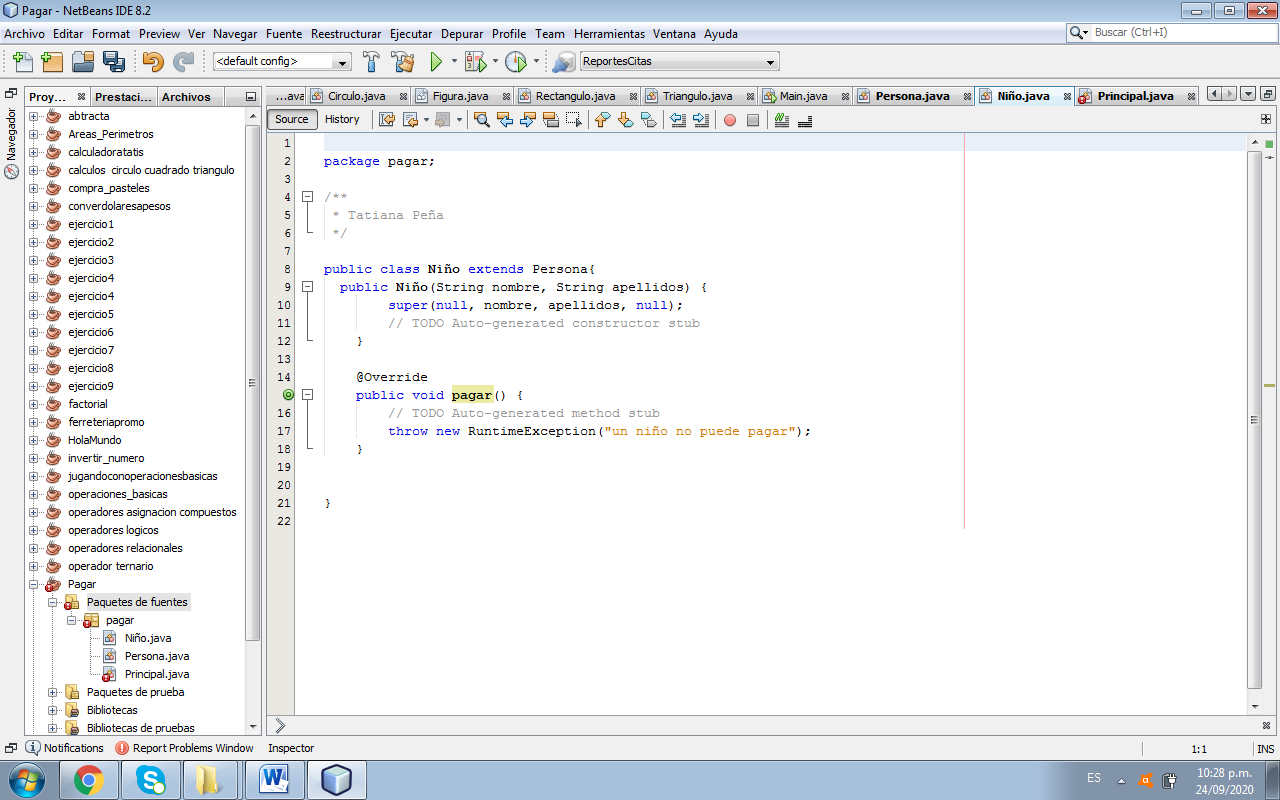
Ejemplo Violación de este principio

Se tiene la clase Persona que dispone de Dni, Nombre, Apellidos y Tarjeta para realizar pagos. El código seria así:

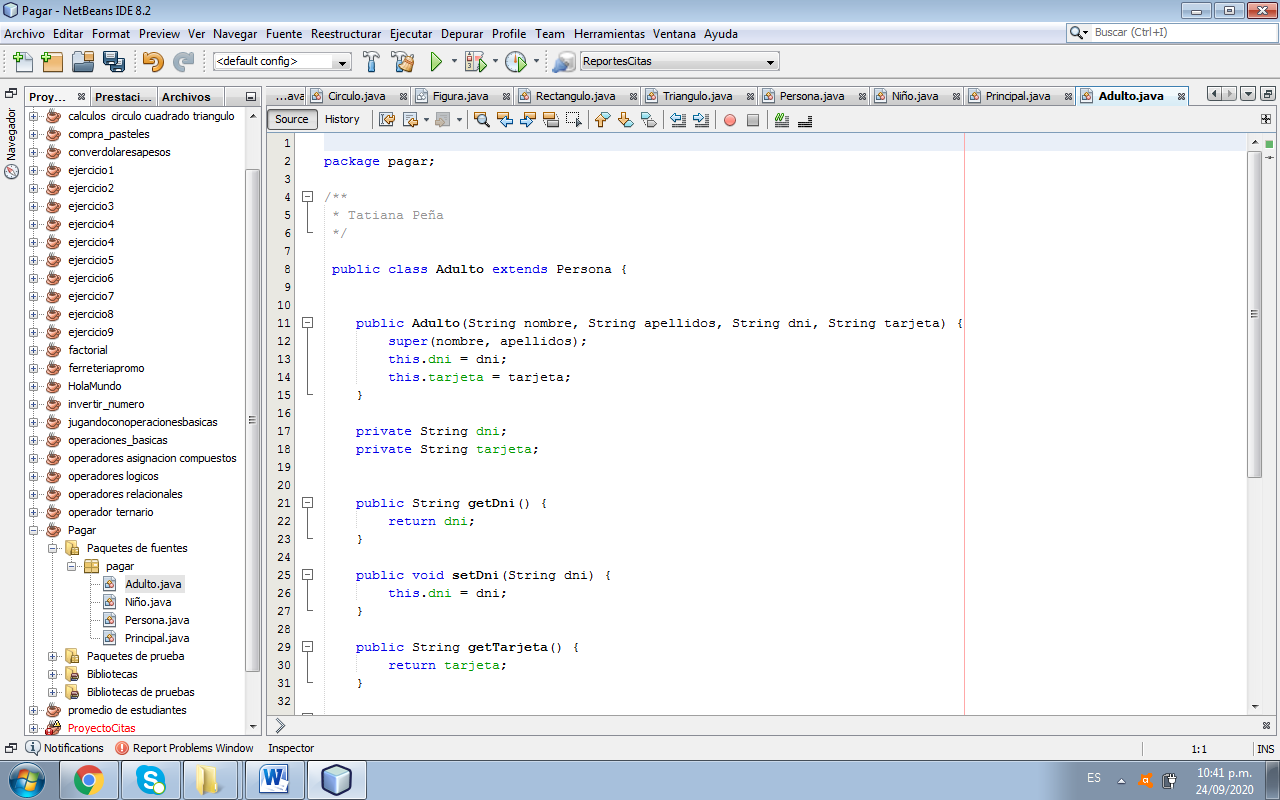


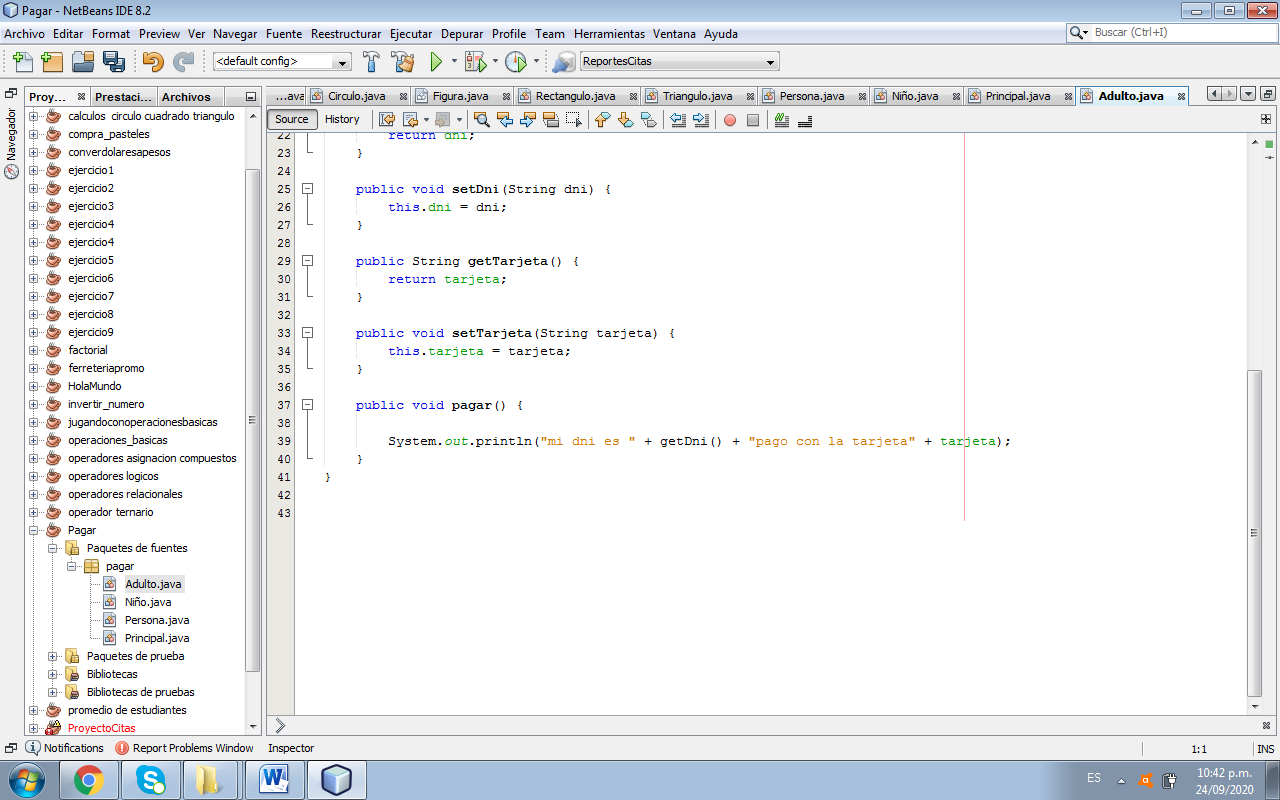


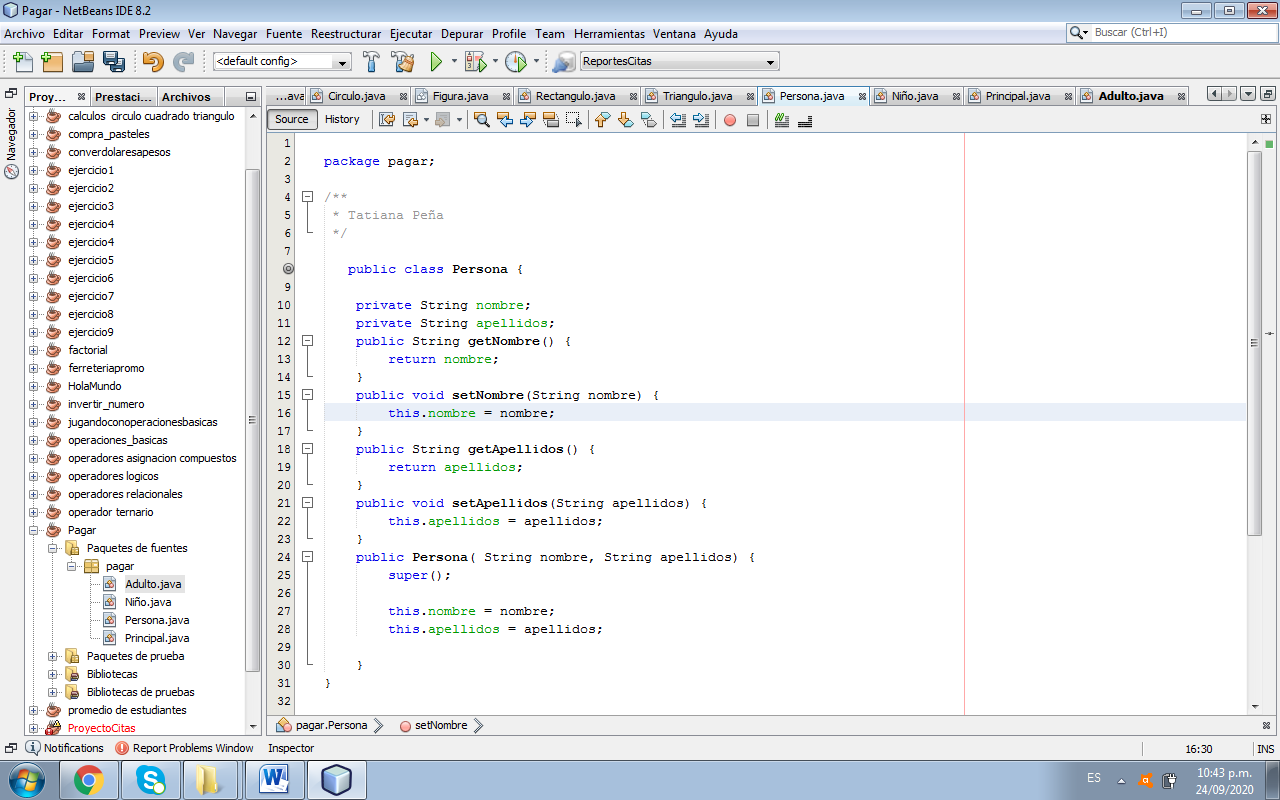
Se hace uso de la herencia para crear una clase hija. En este caso se va a heredar la clase Niño ya que un niño “es una Persona” por lo tanto es correcto usar la herencia ya que estamos ante una relación de categorización o ¿quizás no? Empieza el problema, el niño no tiene Dni , no tiene tarjeta y no puede pagar nada. Sin embargo sí es un tipo de Persona y cumple con la relación “is a. Lo que está sucediendo es que No se está aplicando el Principio de Substitución de Liskov. No se puede substituir la clase padre (T) por la clase hija (S) en muchos lugares del programa sin que esto implique un cambio de comportamiento importante. Para que la clase Hija sea simplemente coherente quedaría así:

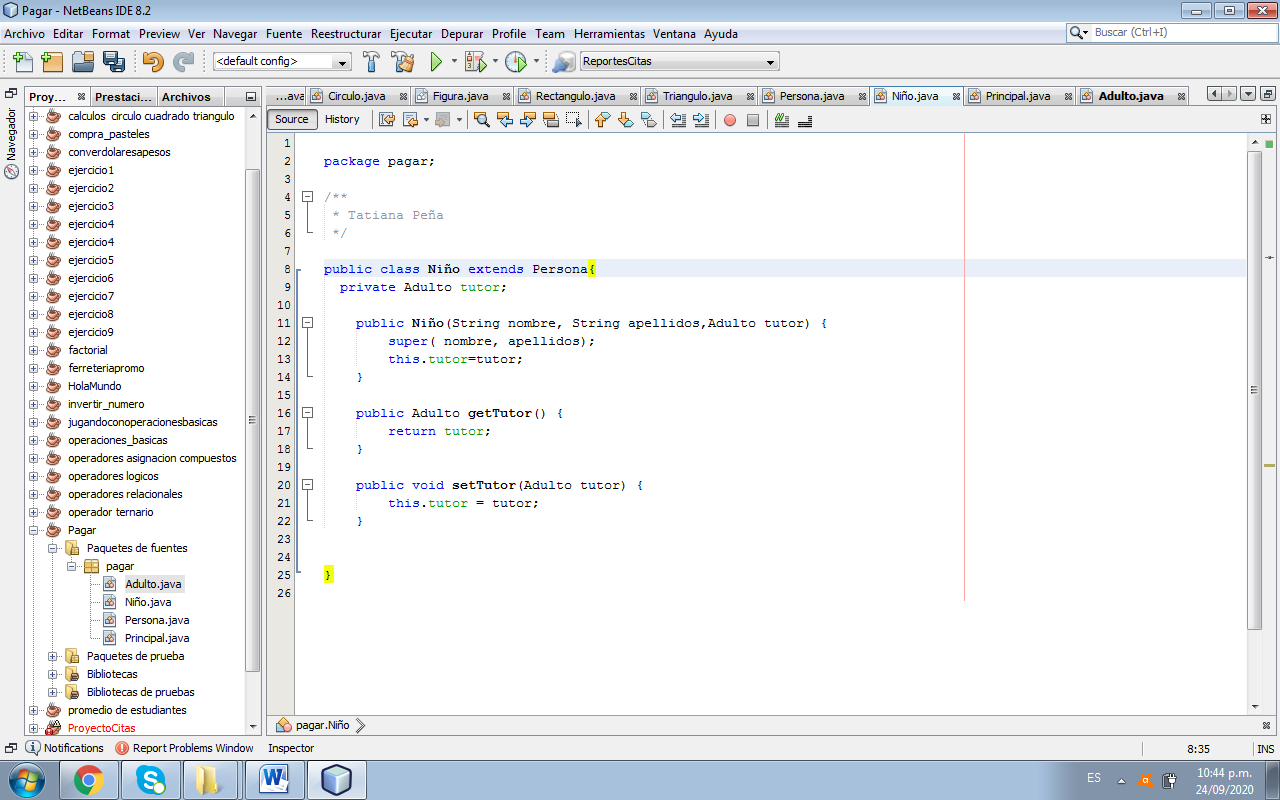


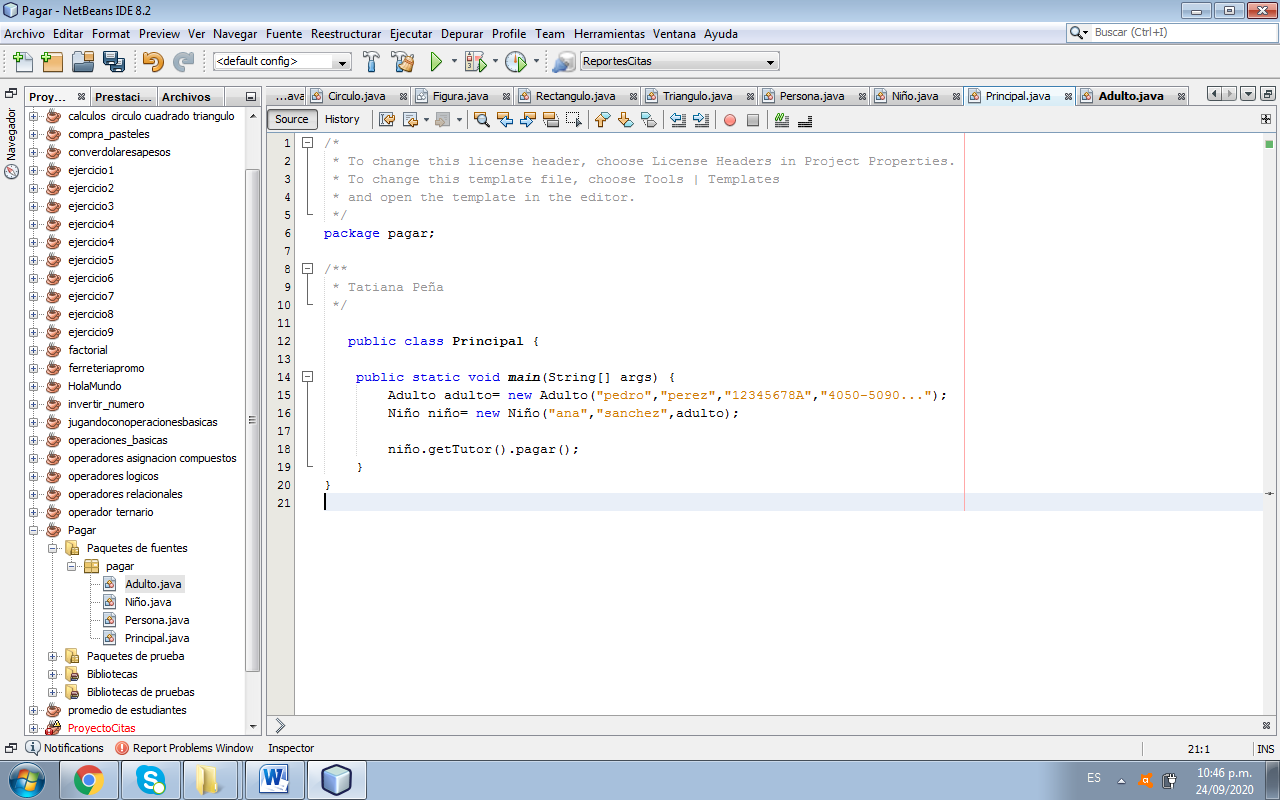
Aplicando de forma estricta el Principio de Substitución de Liskov rediseñando la jerarquía de clases. Como el niño es una Persona ya que siempre tiene nombre y apellidos. Es la clase Adulto la que incorpora el Dni y la tarjeta para pagar. De esta forma todo es más reutilizable. Si se quiere que el niño pueda pagar algo lo hará delegando en la clase Adulto que es la que puede hacerlo con la figura de tutor así:



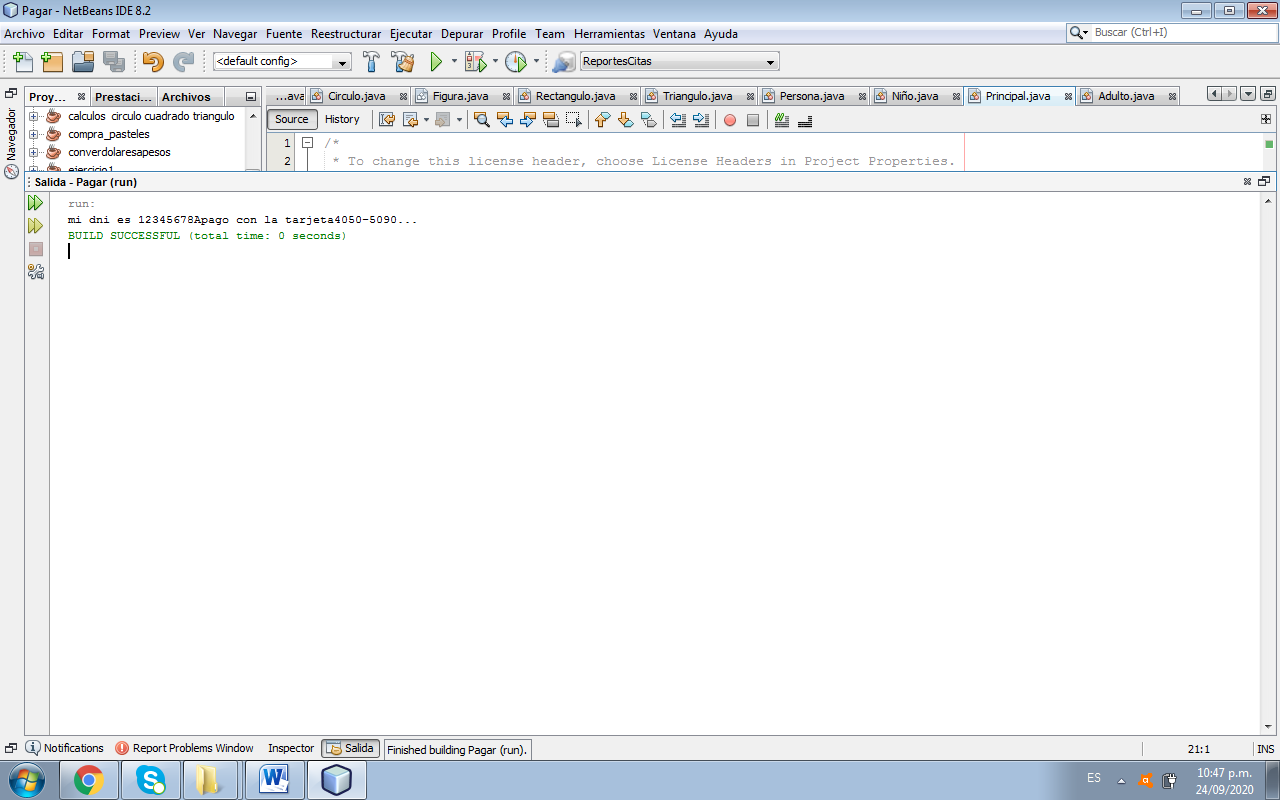








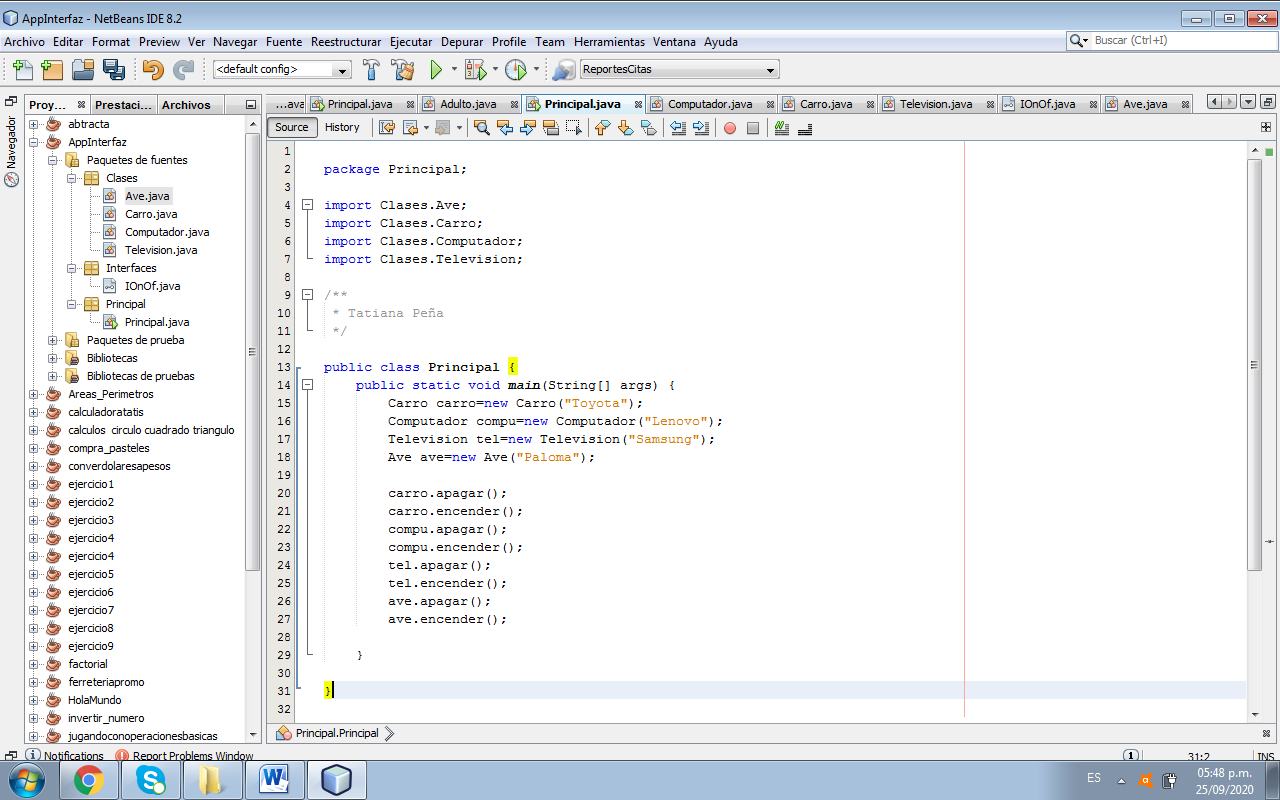
Compilando

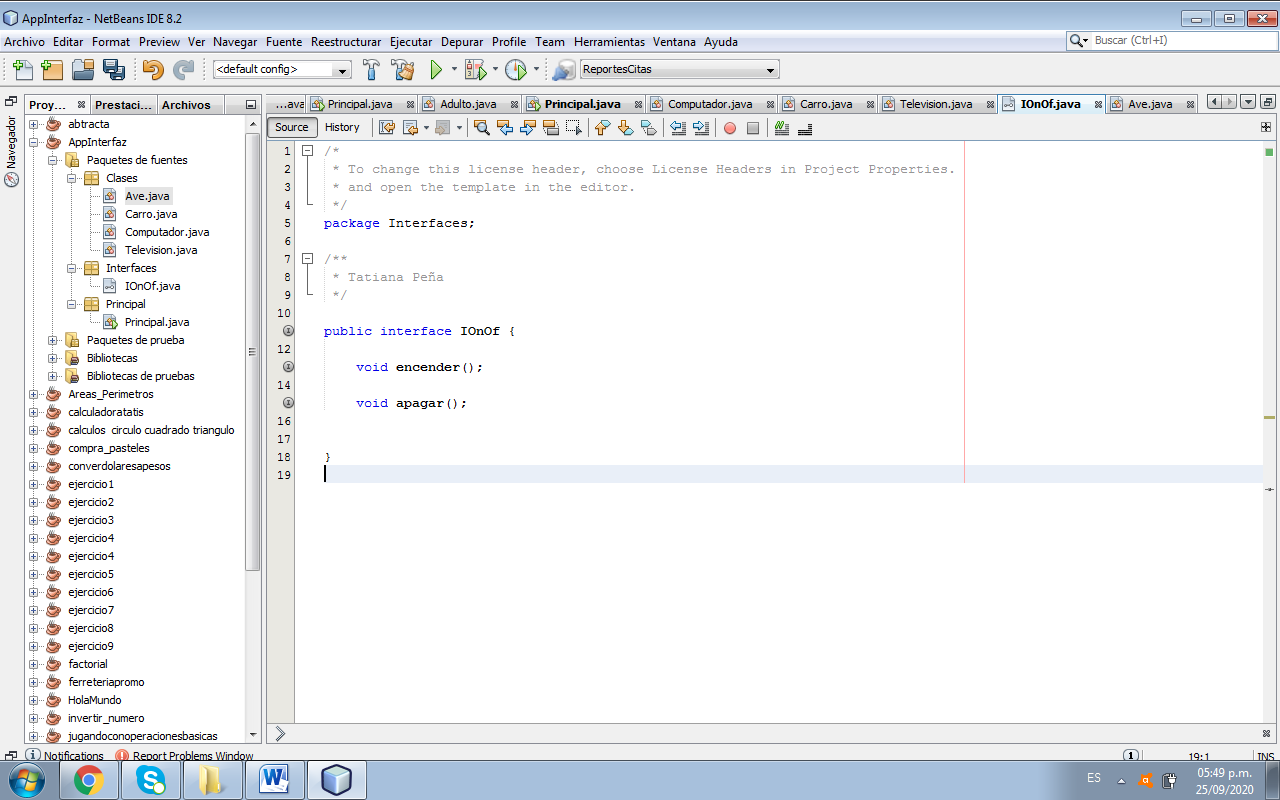


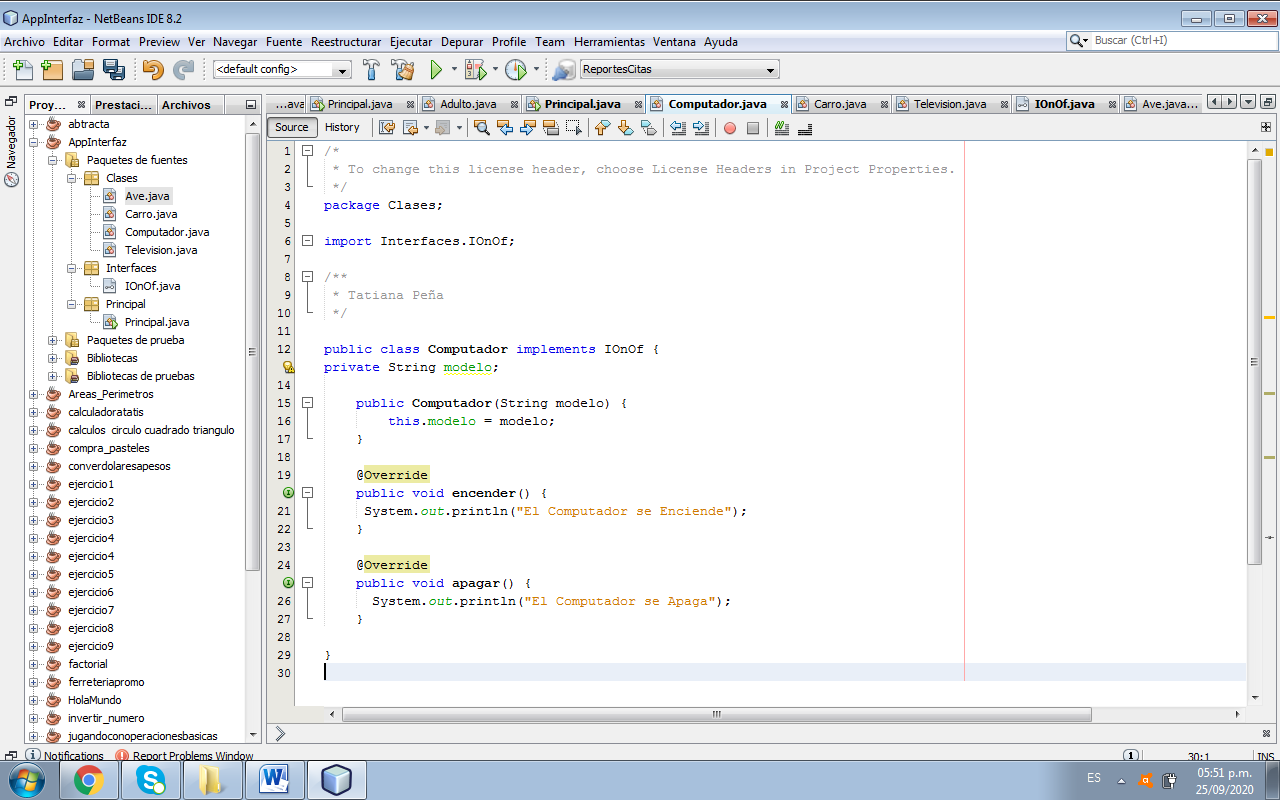
**Principio de Segregación de Interfaces (Interface Segregation Principle),** que dice que se han de utilizar interfaces con propósito específicos, o sea que tengan responsabilidades únicas y que se piensen bien y no se hagan interfaces grandes. Mediante la segregación de interfaces, el planteamiento del diseño otorga una mayor cohesión al sistema, lo que se traduce, por una parte, en un menor coste de mantenimiento, y por otra, en un menor riesgo de errores y una mejor localización de los mismos.

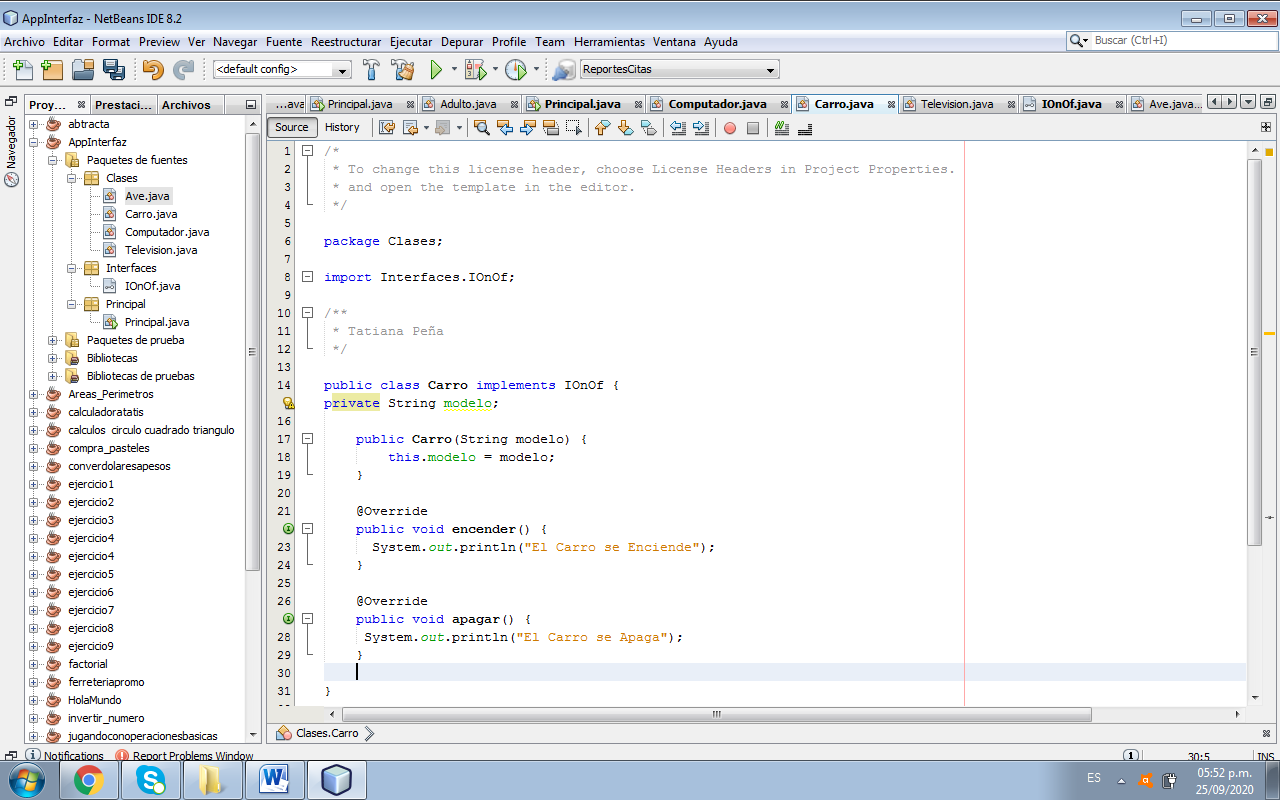
Un ejemplo que está violando este principio.

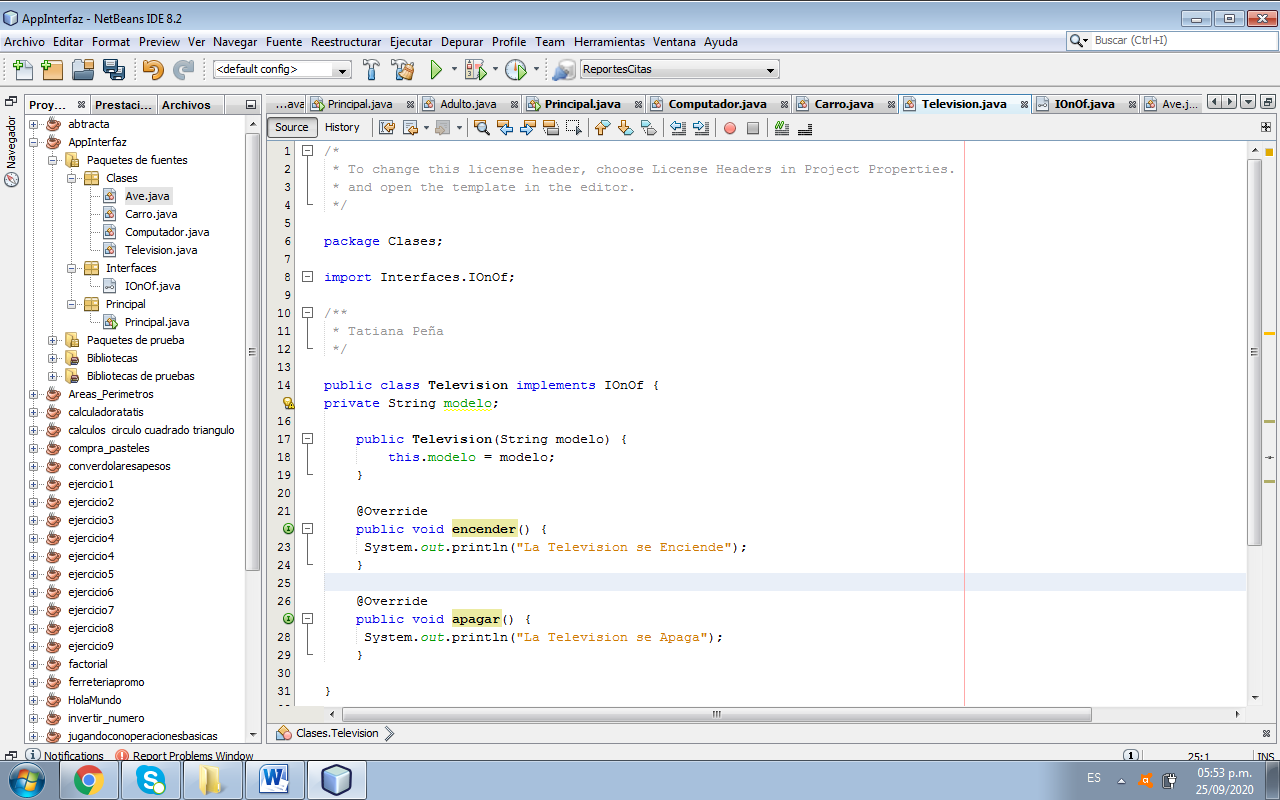
Se hace un programa que tiene una interfaz llamada IOnOf que tiene los métodos apagar y prender, y unas clases Carro, Televisión, Computador, Ave y la case principal para compilar desde ella; el principio se viola porque a la clase Ave no le aplican los métodos. Aquí el código

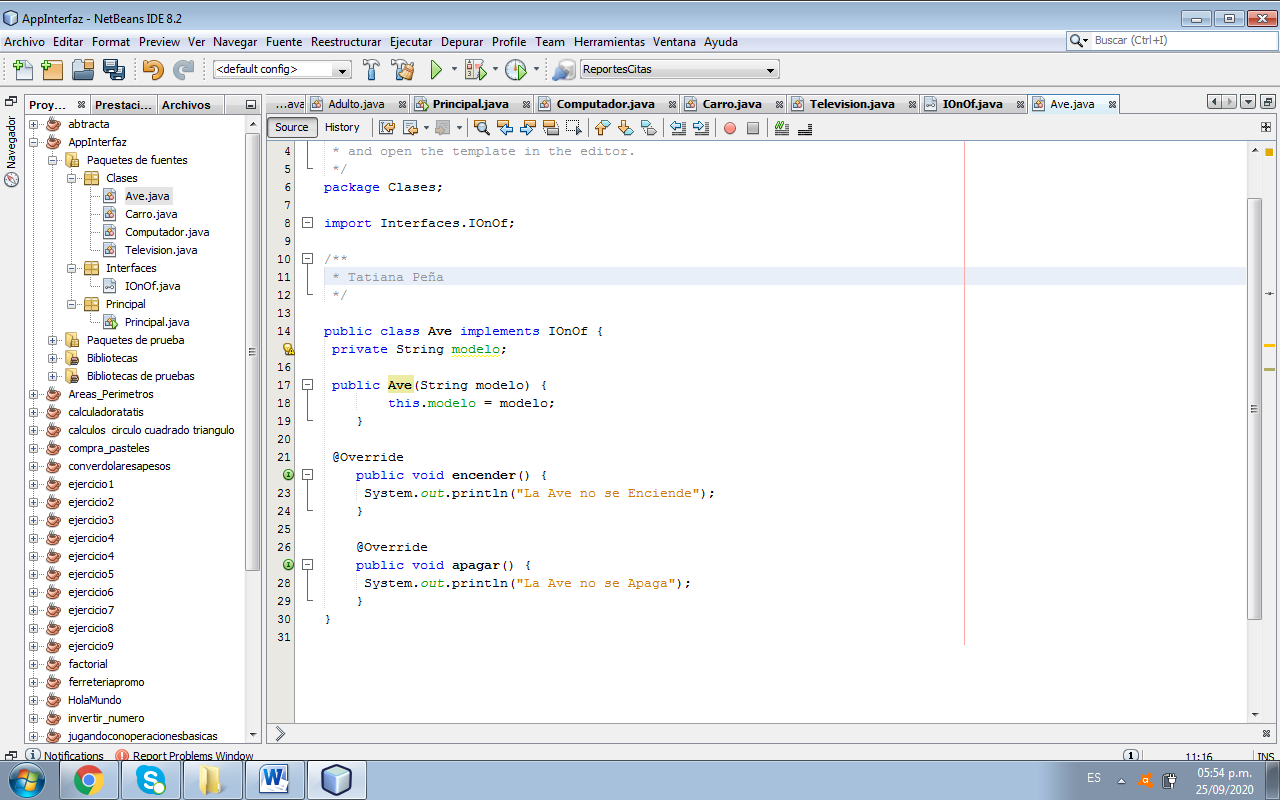




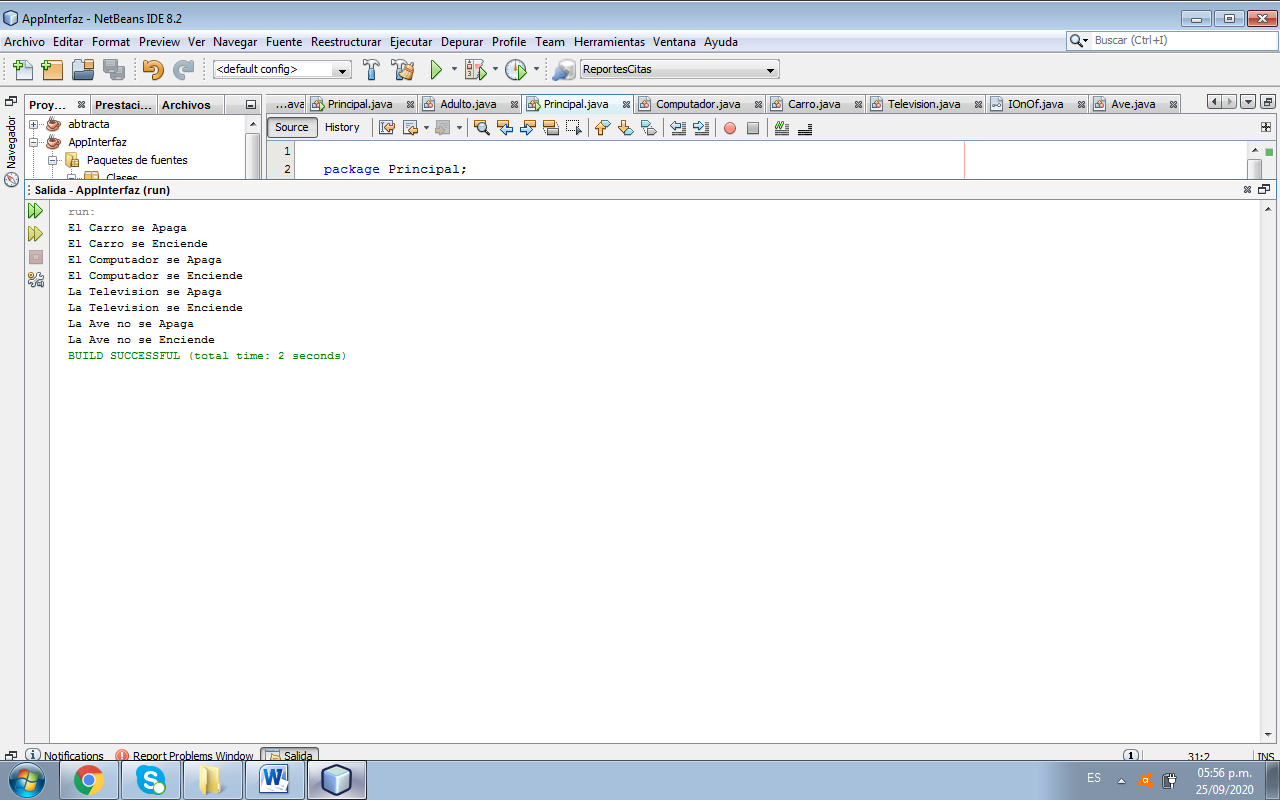




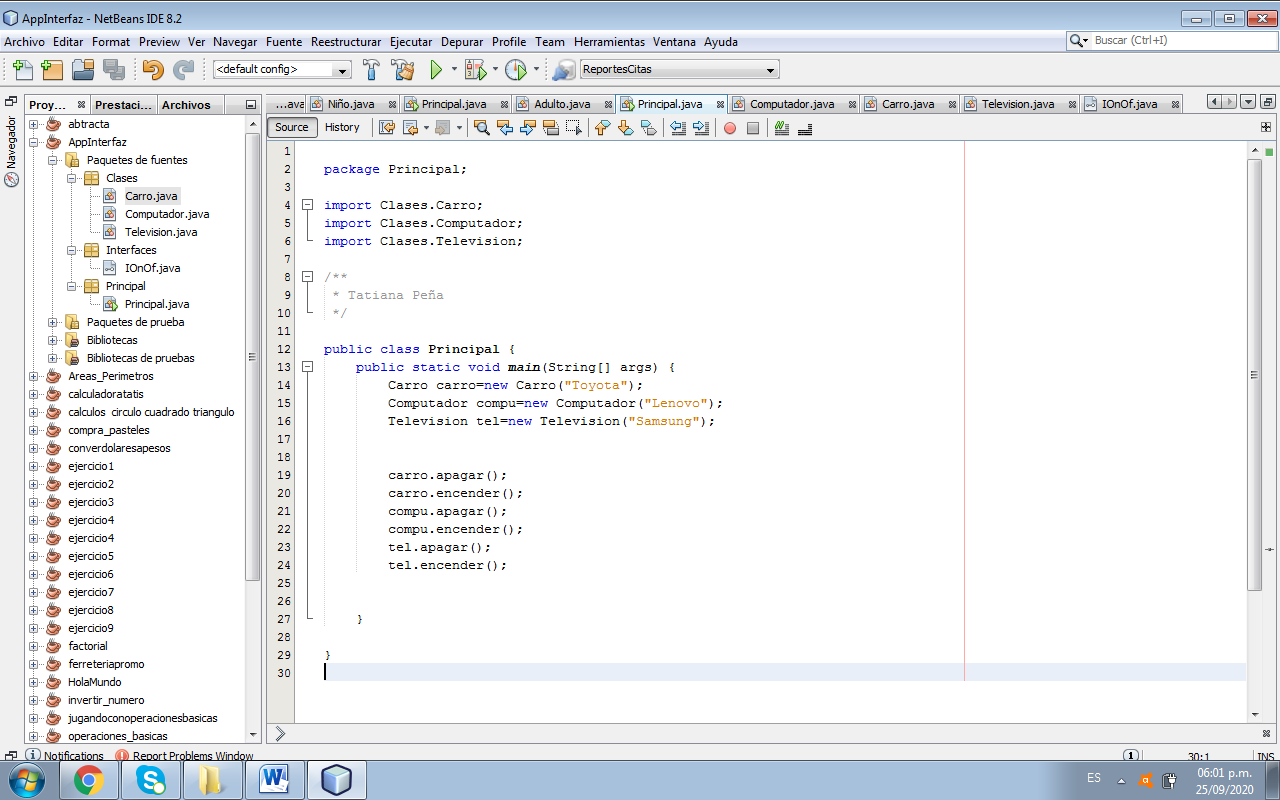




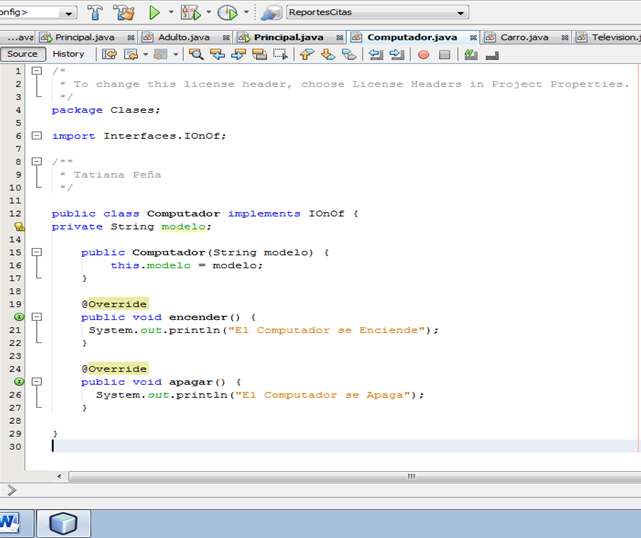
Al ejecutar el programa quedaría así:

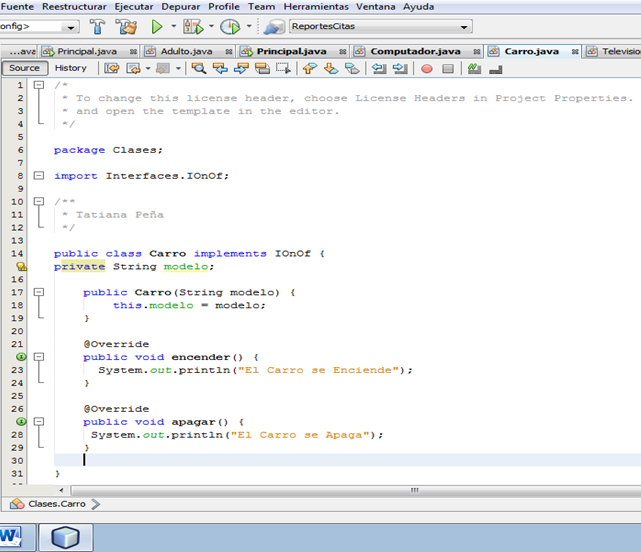


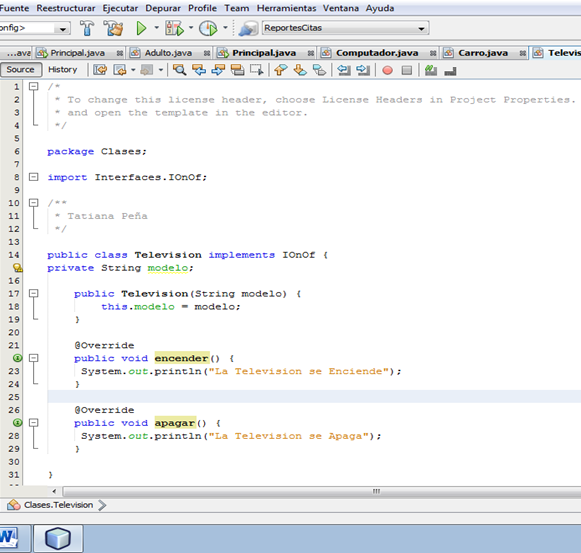
Aplicación correcta del principio de Interfaces será así:



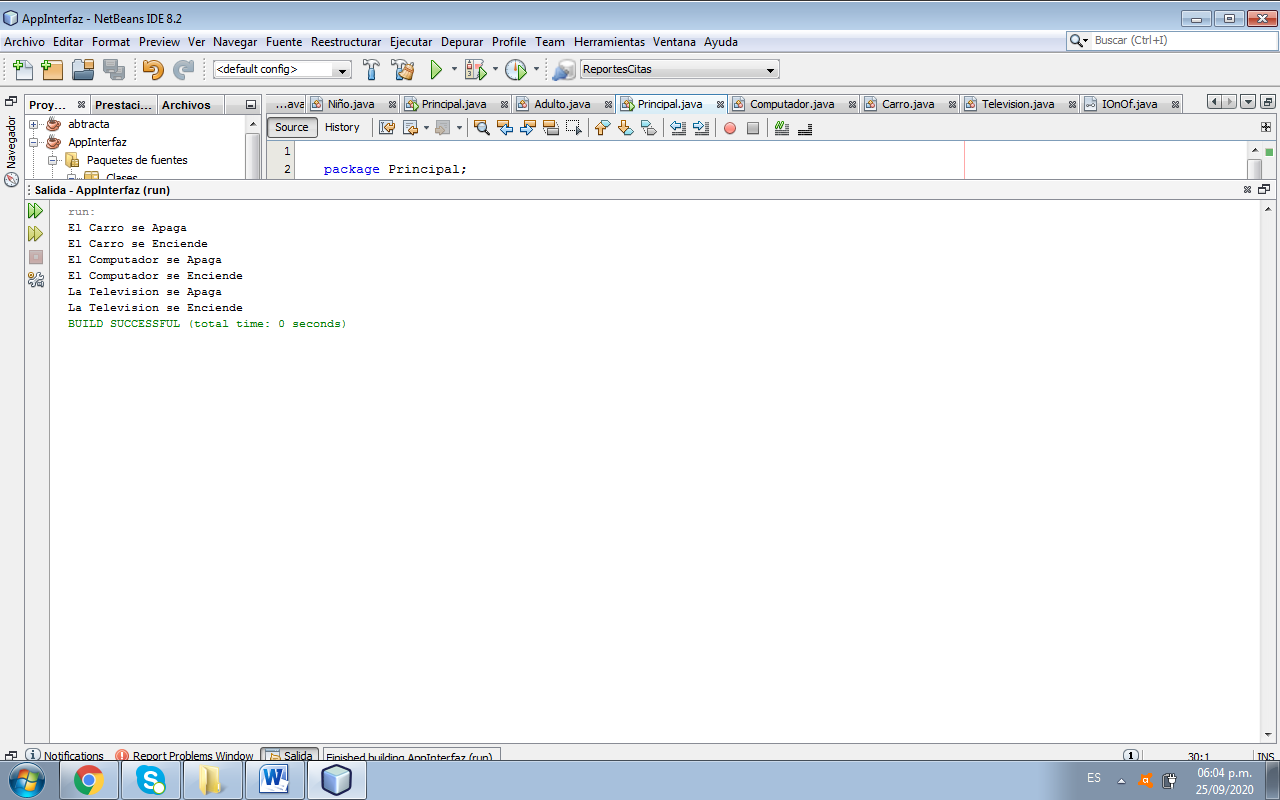








Al ejecutar quedaría así:



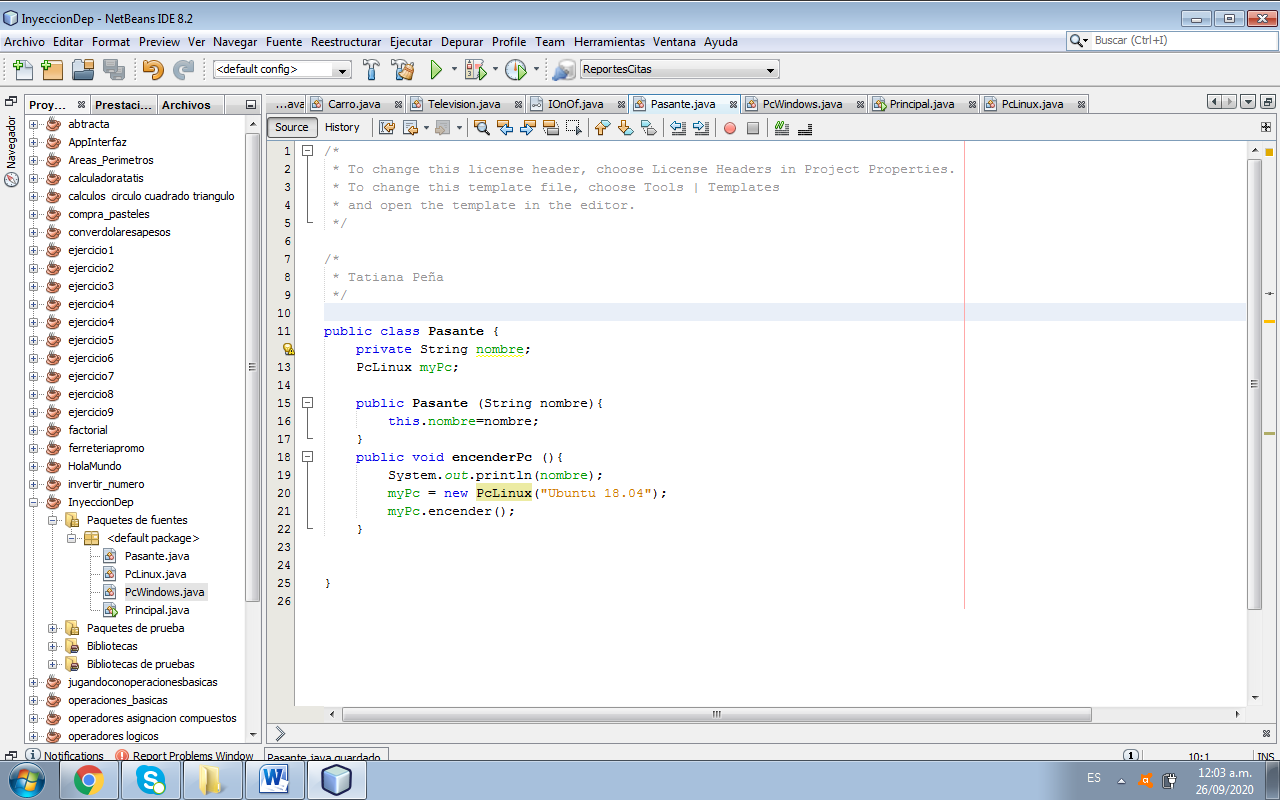
**Dependency inversión principle o Principio de inversión de dependencia**

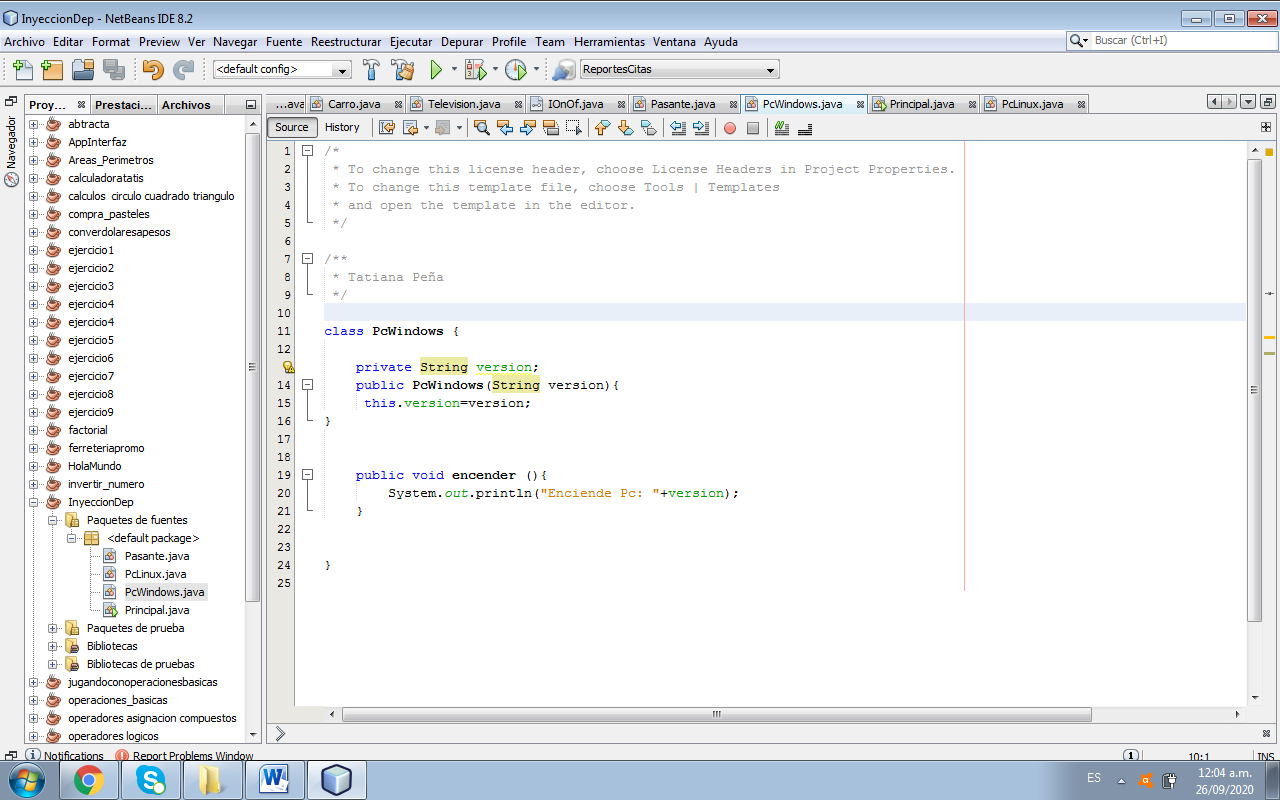
Básicamente lo que nos dice este principio es que: Las clases de alto nivel no deberían depender de las clases de bajo nivel. Ambas deberían depender de las abstracciones.

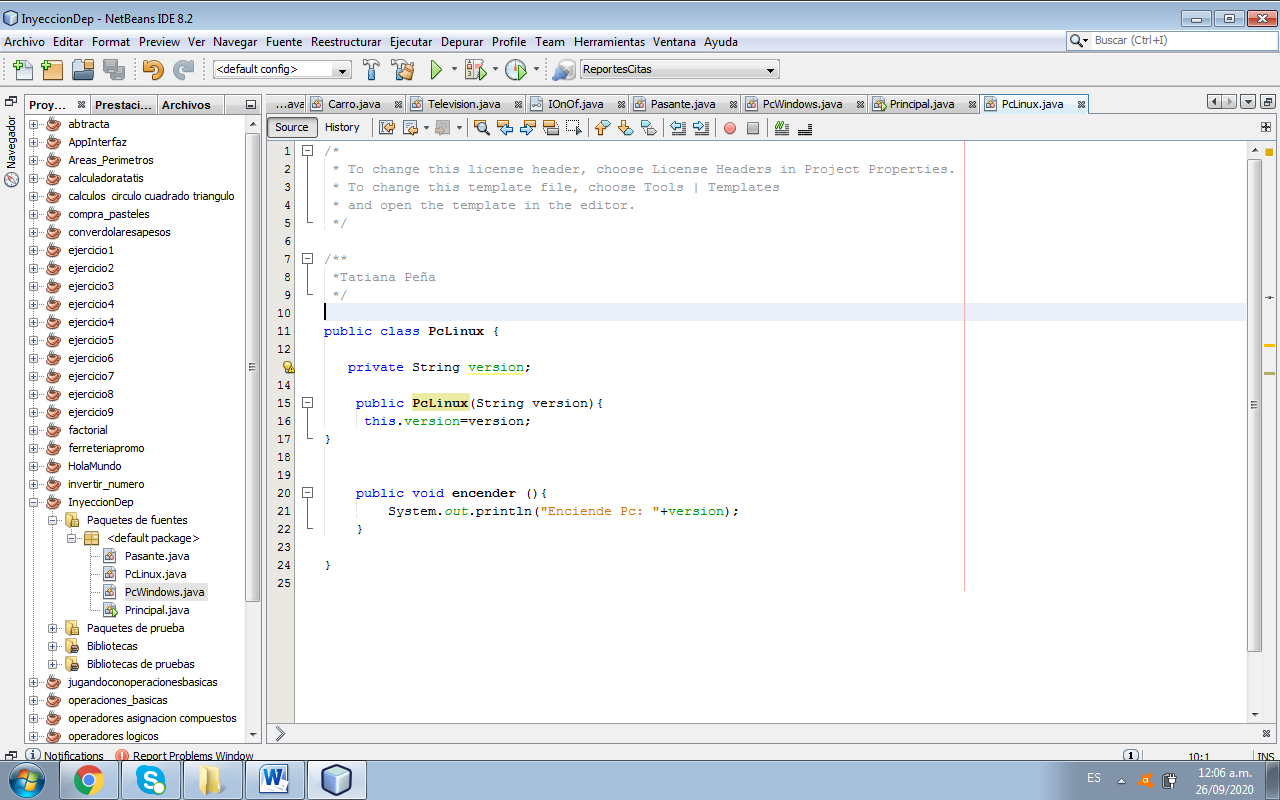
Las abstracciones no deberían depender de los detalles. Los detalles deberían depender de las abstracciones. Esto no es más que extraer responsabilidades a un componente para delegarlas en otro, estableciendo un mecanismo a través del cual el nuevo componente pueda ser cambiado en tiempo de ejecución. Es conveniente no confundir Inyección de dependencias (DIP) con Inversión de Control (IoC).

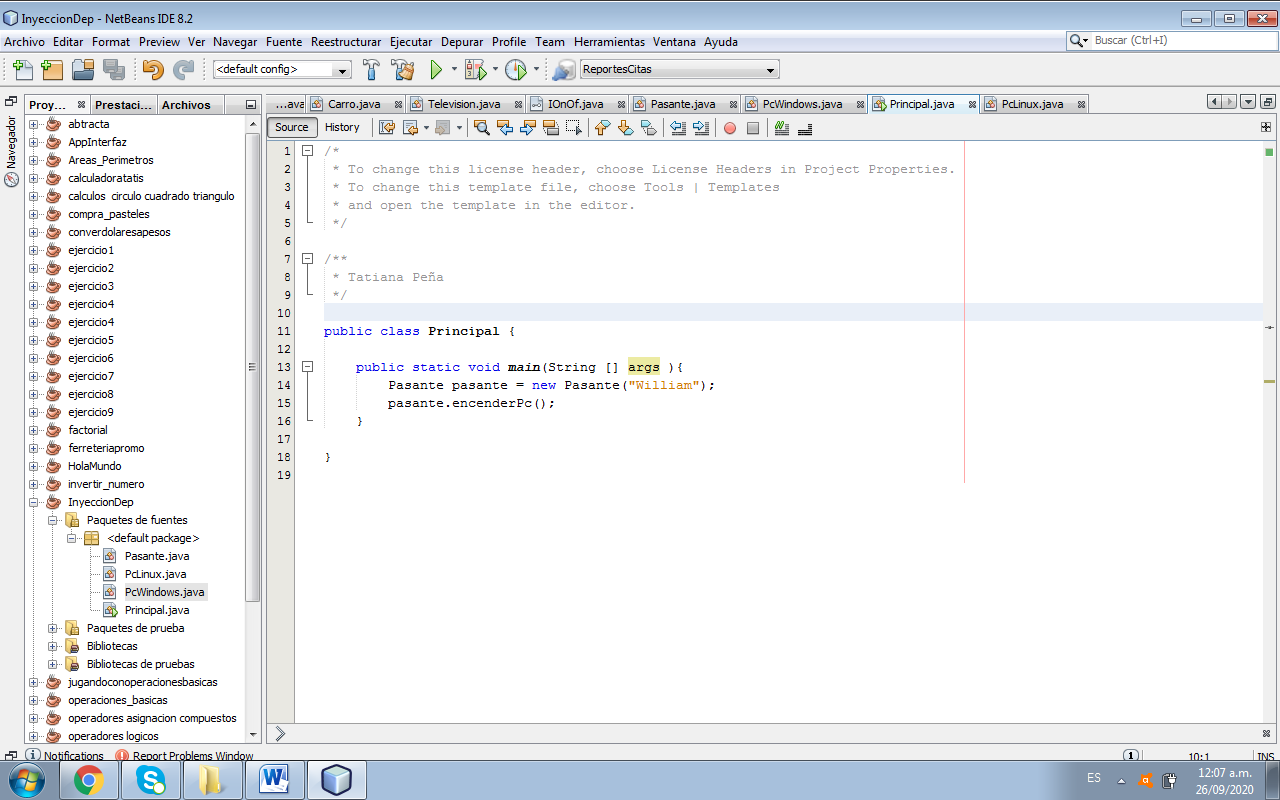
Ejemplo de violación de este principio

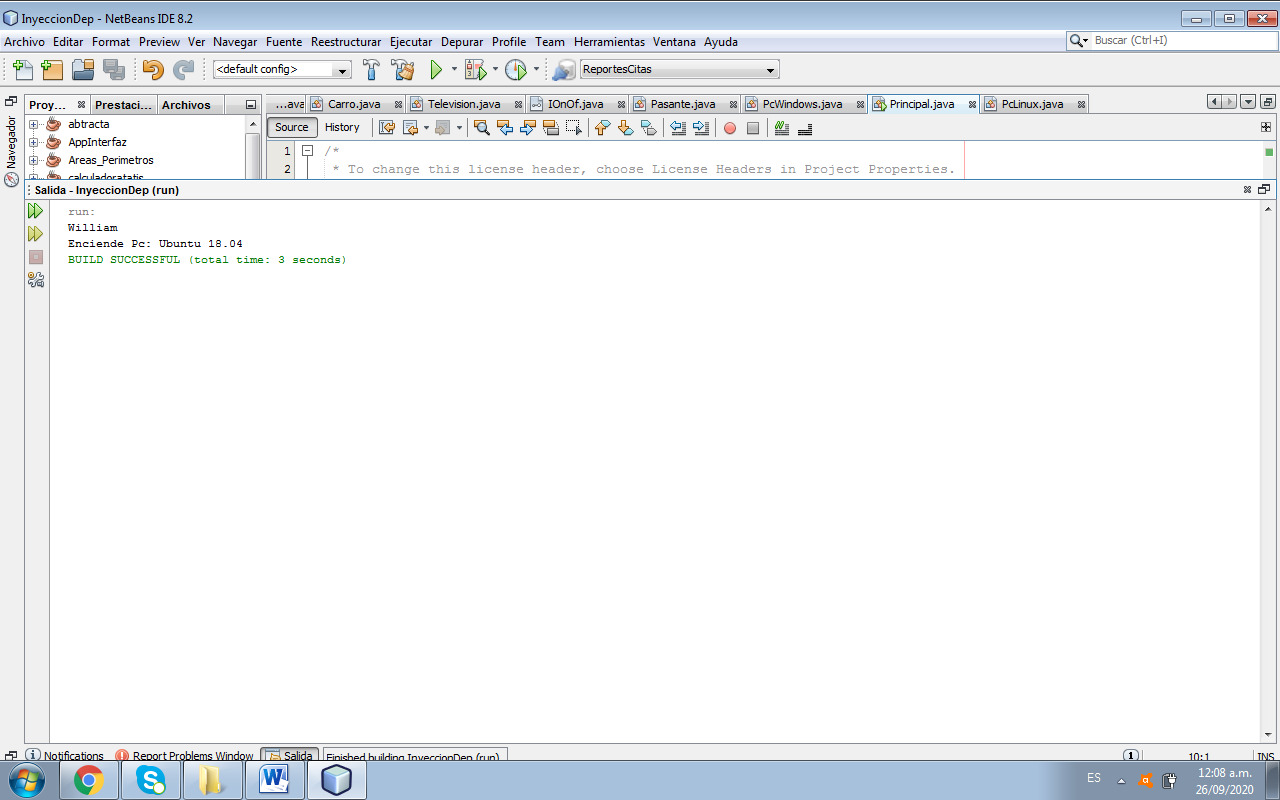
Se hace un programa que tiene las clases Pasante, PcWindows, PcLinux, principal; no se cumple el principio por el alto acoplamiento ya se está modificando la clase Pasante así:



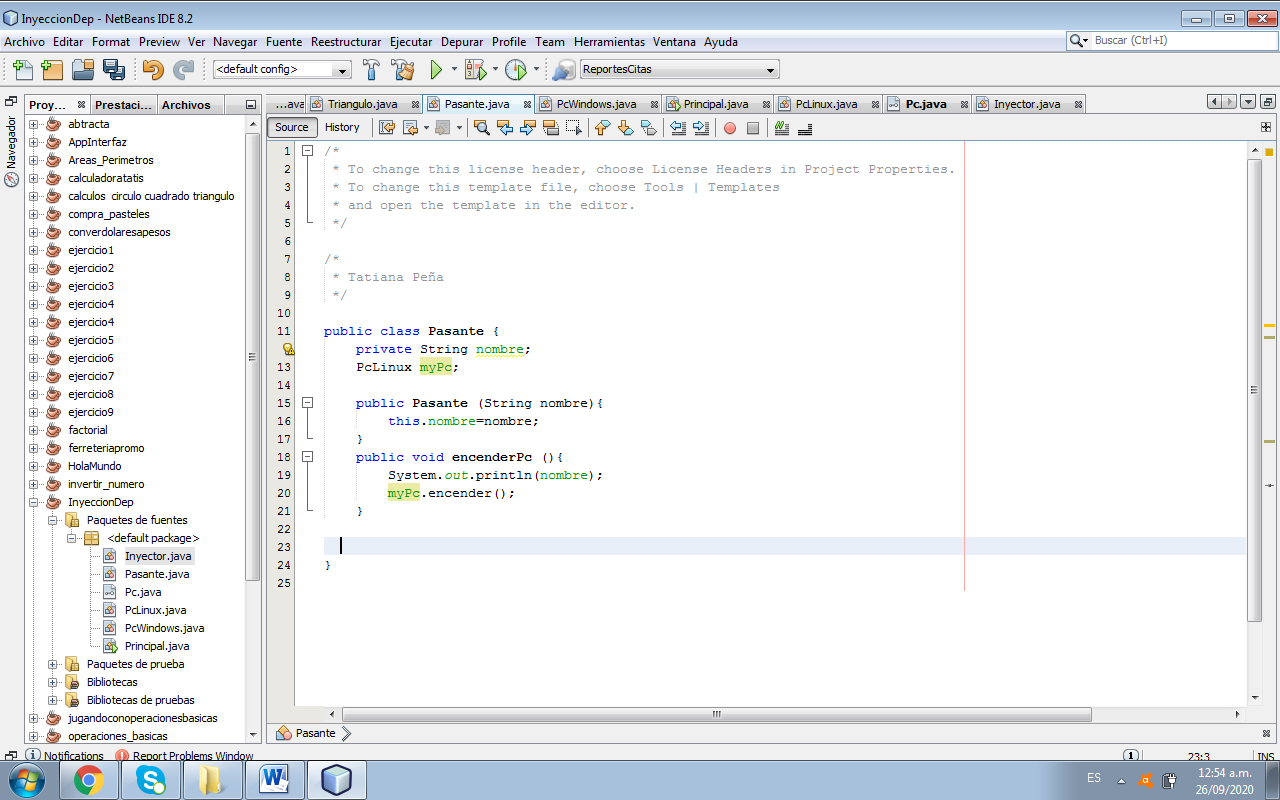


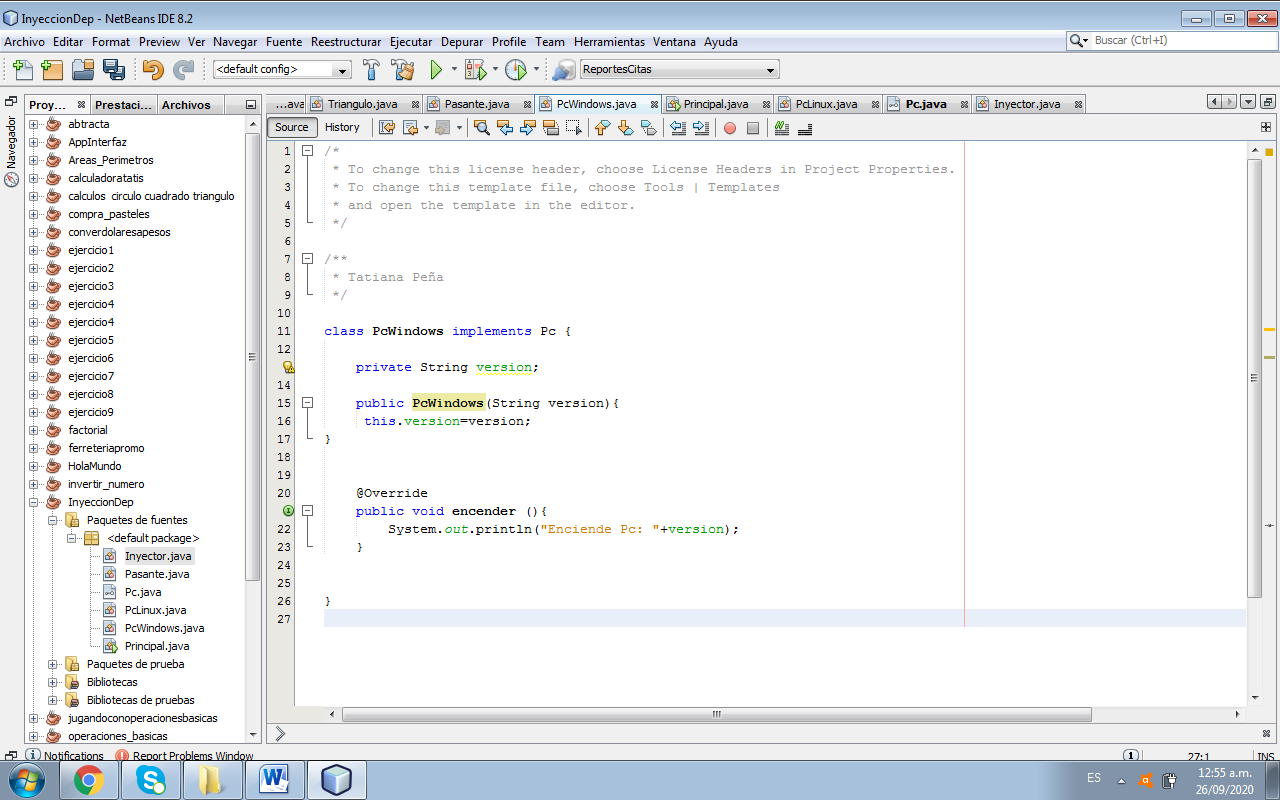


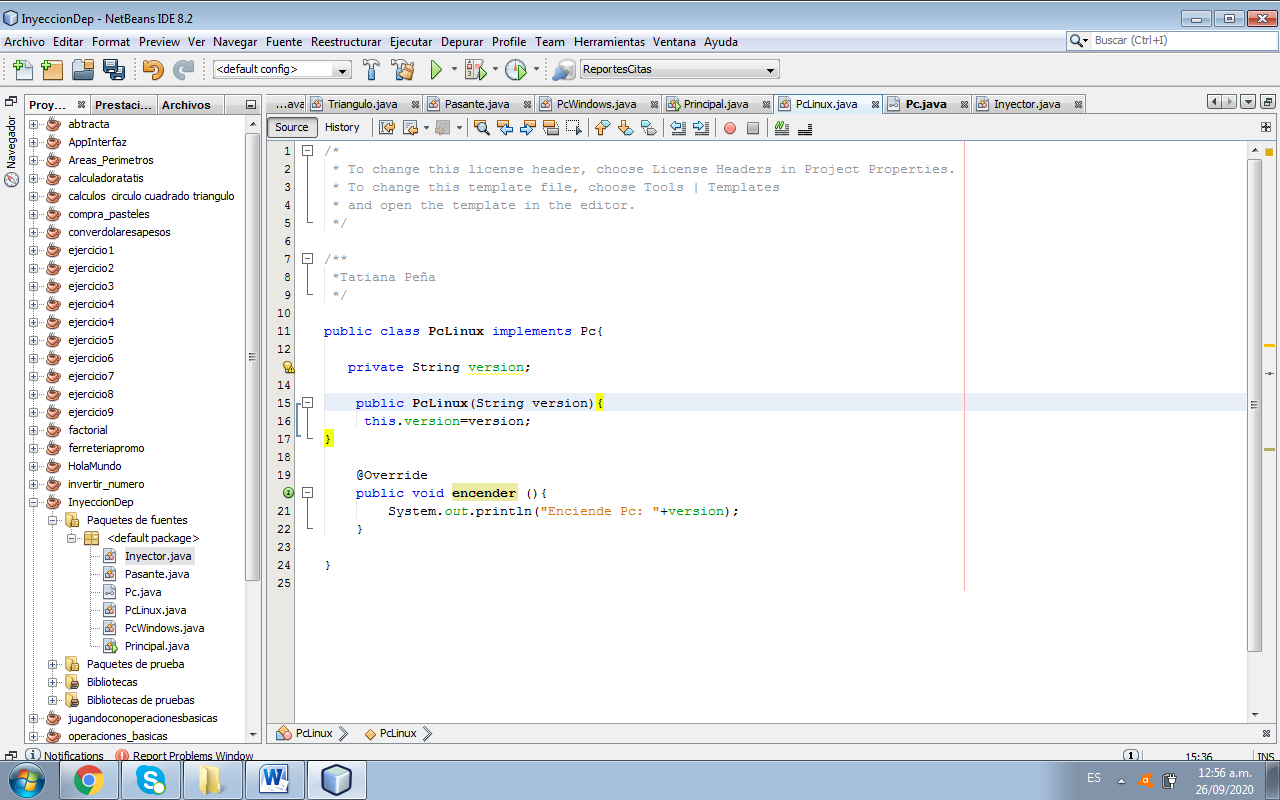


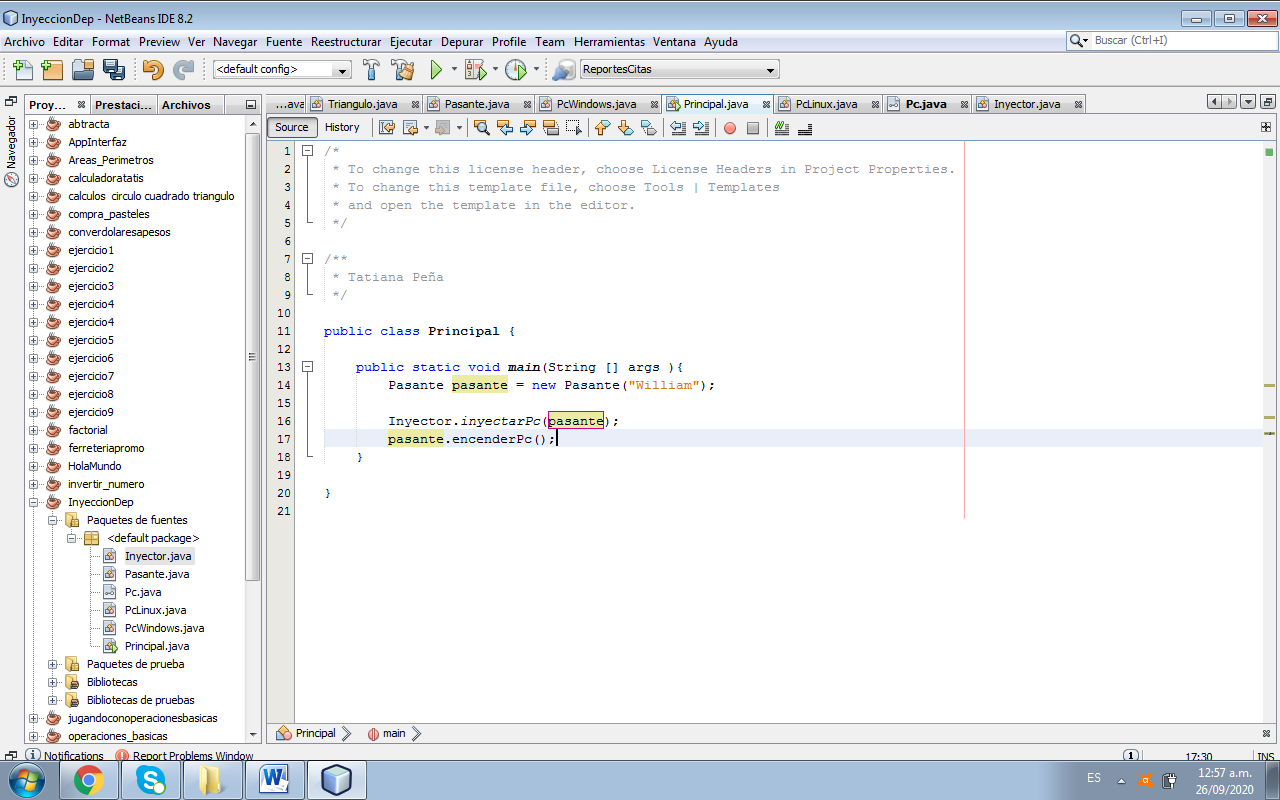


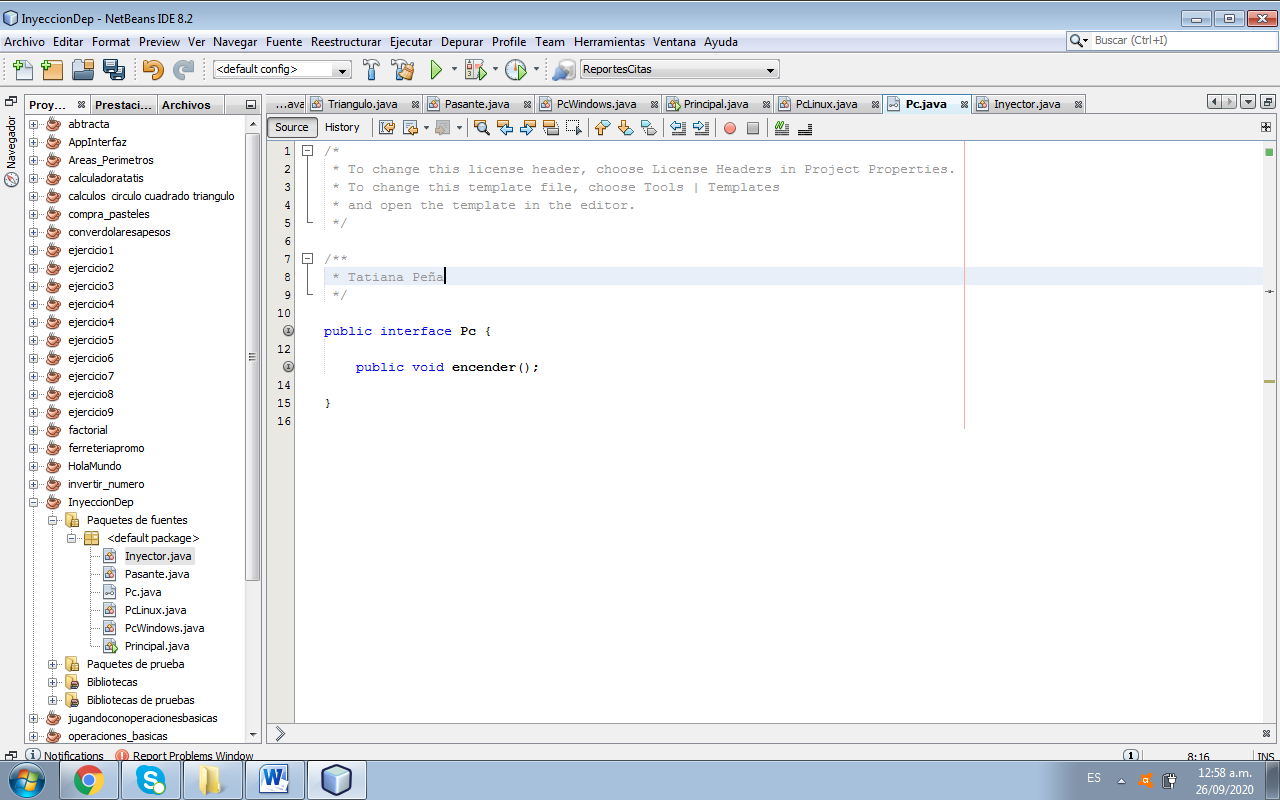
Ejemplo Respetando este principio se aplica la inyección de dependencias

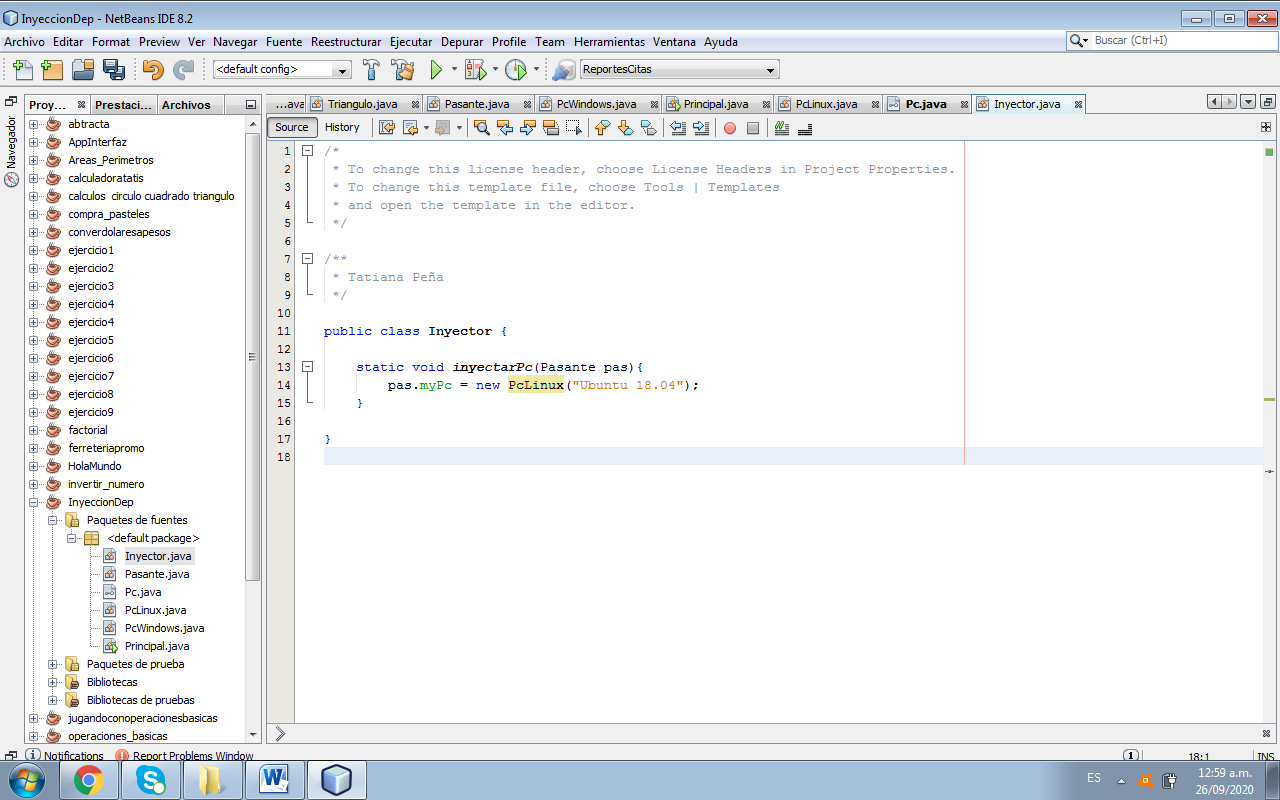


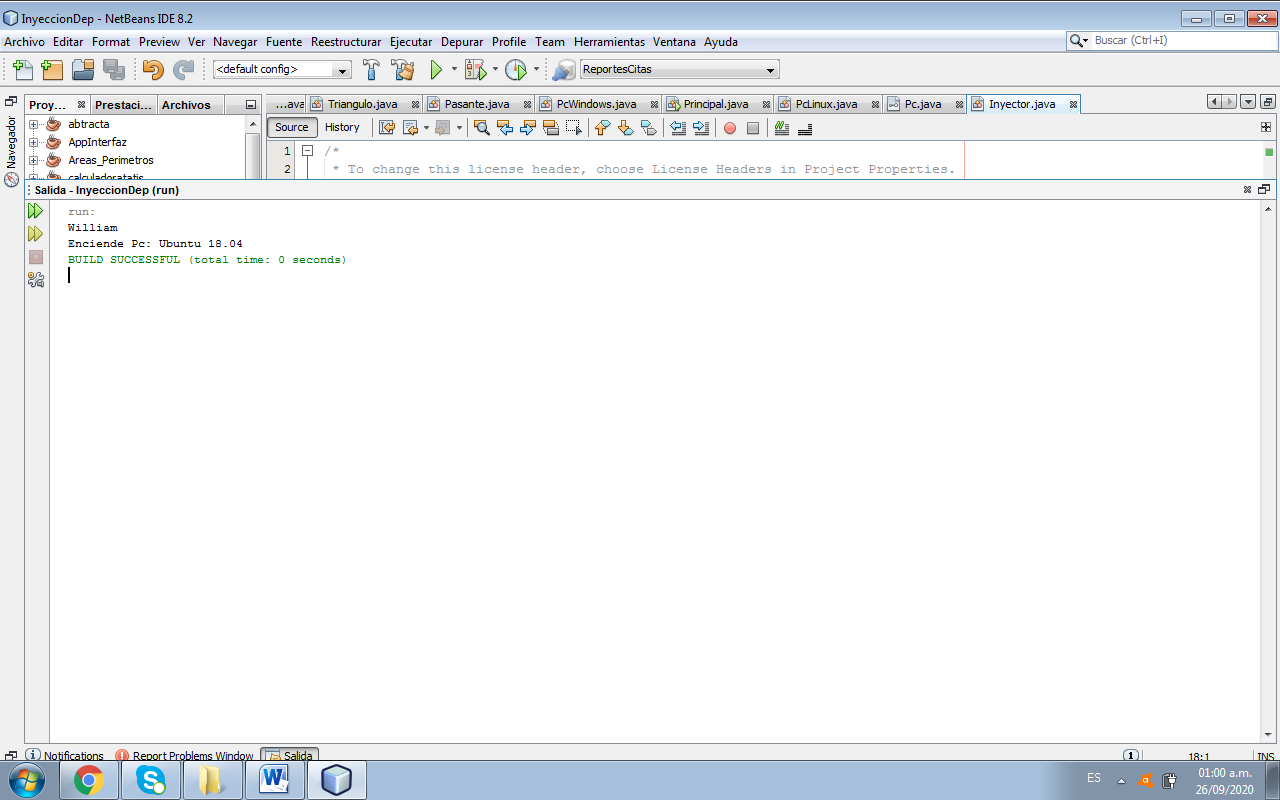












<https://jrrn.github.io/grasp> Patrones Grasp

<https://www.youtube.com/watch?v=RRegIKu-z3k>

<https://www.youtube.com/watch?v=2HRgYWqb8KY> ejemplo segregación interfaces

<https://enmilocalfunciona.io/principios-solid/> revisar

<https://docs.microsoft.com/en-us/learn/paths/intro-to-vc-git/>

<https://devexperto.com/principio-de-sustitucion-de-liskov/>

<https://www.arquitecturajava.com/el-principio-de-substitucion-de-liskov/>

<https://tech.tribalyte.eu/blog-solid-open-closed>

<https://joanluk.wordpress.com/2015/08/09/principio-abiertocerrado-openclosed/>

<http://blogjavacartagena.blogspot.com/2014/03/clase-abstracta-en-java.html>

<https://desarrolloweb.com/articulos/principio-reponsabilidad-unica-II-dotnet.html>

<https://openwebinars.net/blog/que-es-solid/>

<http://estructura-de-datos-oo.blogspot.com/2013/09/calcular-areas-y-perimetros-triangulos.html>

<https://ingenieria.udistrital.edu.co/course/view.php?id=1123>

<https://www.arquitecturajava.com/el-patron-de-inyeccion-de-dependencia/>