UNTDF
Universidad Nacional de Tierra del Fuego, Antártida e Islas del Atlántico Sur

Principios de Diseño SOLID

Principios que orientan el reuso en OO

Responsabilidad única – Separación de cuestiones
Una clase debe tener una única razón para cambiar

Principio Open/close
Las entidades de Software deben estar abiertas para extensión pero cerradas a modificaciones.

Sustitución (Liskov)
Si un programa está usando una clase base, luego la referencia a la clase base puede ser reemplazada con una clase derivada sin cambiar afectar la funcionalidad general.

Inversión de Dependencia
Los módulos de alto nivel no deben depender de los módulos de bajo nivel. Ambos deben depender de abstracciones.
Las abstracciones no deben depender de detalles. Los detalles deben depender de las abstracciones.

Segregación de Interface
Los clientes no deben ser forzados a depender de interfaces implementadas que no usan.

UNTDF
Universidad Nacional de Tierra del Fuego, Antártida e Islas del Atlántico Sur

Principios de Diseño SOLID

Responden a Problemas en el diseño

Rigidez: El sistema es difícil de cambiar, porque cada cambio fuerza muchos otros en distintas partes del sistema.
Fragilidad: Los cambios causan que el sistema falle en lugares que no están conceptualmente relacionados con la parte que cambió.
Inmovilidad: Es difícil separar el sistema en componentes que puedan ser reusados en otros sistemas.
Viscosidad: Aparece en proyectos en los cuales el buen diseño del SW es difícil de preservar ante cambios realizados
Complejidad innecesaria: Contiene elementos que no son frecuentemente utilizados
Repetición innecesaria: El diseño contiene estructuras repetitivas que podrían ser unificadas bajo una abstracción simple
Opacidad: El diseño es difícil de leer y comprender, no expresa claramente su meta.

Principios de Diseño SOLID

Principios de Diseño OO: SOLID

- **SOLID es un acrónimo para 5 principios de diseño de objetos,**
- Se utiliza en base a los "primeros cinco principios" de diseño y programación enunciados por Robert Martin en el año 2000.
- Estos principios se aplican todos juntos, y ayudan a que los sistemas sean más mantenibles y fáciles de extender en el tiempo.
- Se pueden aplicar los principios SOLID en código existente como guía para realizar refactorización y durante la codificación de sistemas nuevos.

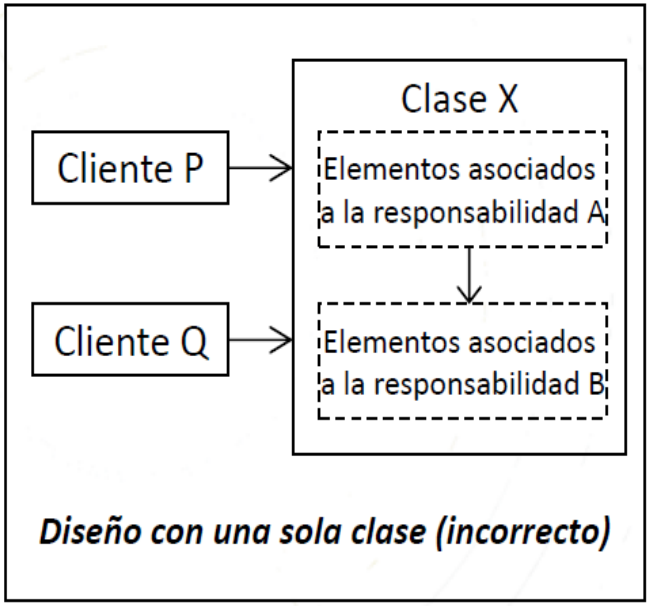
UNTDF
Universidad Nacional de Tierra del Fuego, Antártida e Islas del Atlántico Sur

Principios de Diseño SOLID

Principio de responsabilidad única

Cada objeto en el sistema deben tener una única responsabilidad

- Finalidad
  - Evitar que el cambio de una responsabilidad en una clase pueda provocar fallos en las demás responsabilidades de la clase
  - Evitar que los clientes de una clase carguen con elementos que no utilizan



Diseño con una sola clase (incorrecto)

UNTDF
Universidad Nacional de Tierra del Fuego, Antártida e Islas del Atlántico Sur

Principios de Diseño SOLID

Principio de responsabilidad única

- Este principio fue llamado Cohesión por metodológos tradicionales
- El Principio de responsabilidad única (Single Responsability Principle-SRP) fue acuñado por *Robert C. Martin.*

- En términos prácticos, este principio establece que:

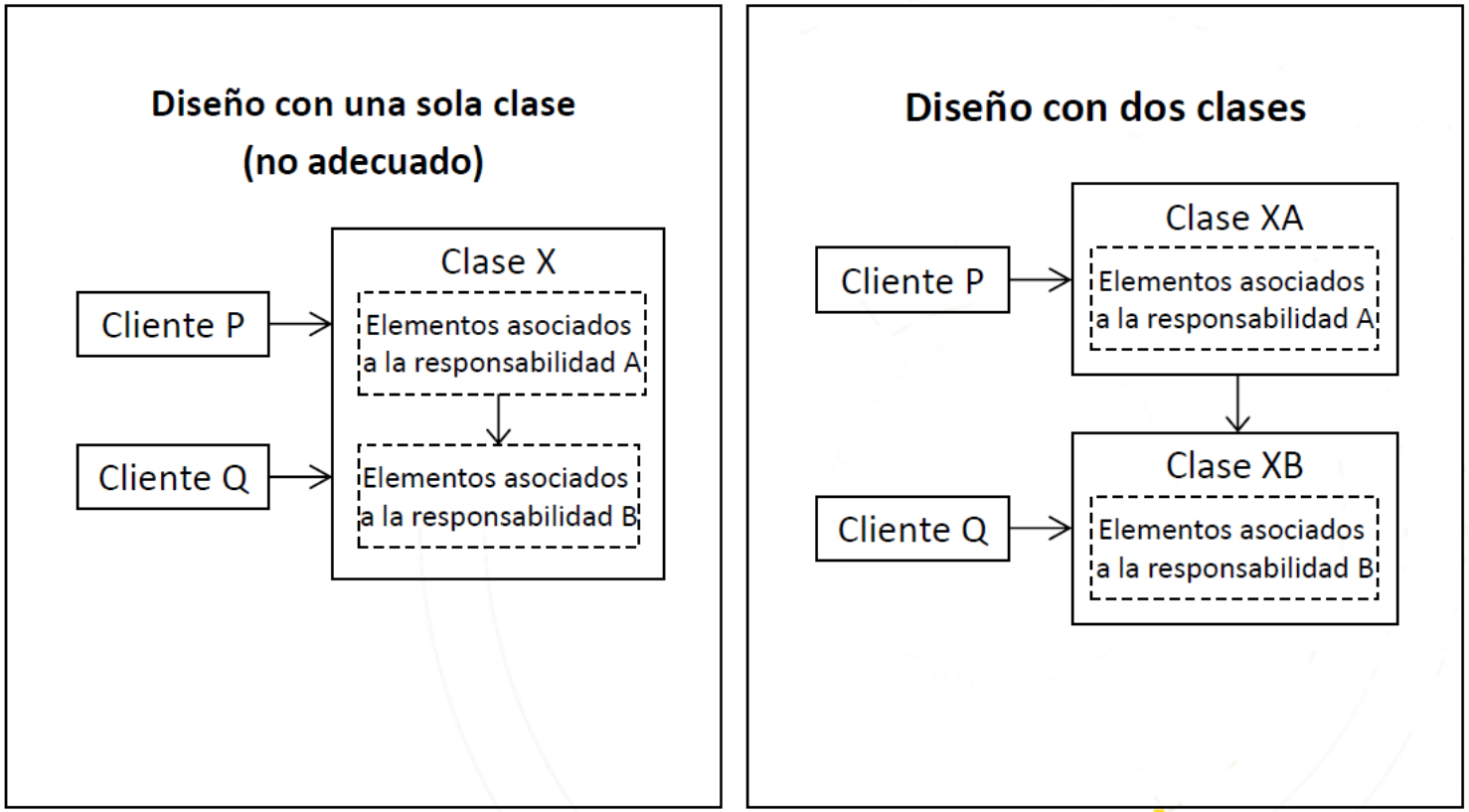
"Una clase debe tener una y solo una única causa por la cual puede ser modificada."

- Cada clase debe ser responsable de realizar una actividad del sistema .
- Si existe más de una responsabilidad, es conveniente agregar otra clase.

UNTDF
Universidad Nacional de Tierra del Fuego,
Antártida e Islas del Atlántico Sur

Principios de Diseño SOLID

Principio de responsabilidad única



UNTDF
Universidad Nacional de Tierra del Fuego, Antártida e Islas del Atlántico Sur

Principios de Diseño SOLID

Principio de responsabilidad única

- Por qué es importante separar las responsabilidades en otra clase?

Porque cada responsabilidad es un vector de cambio (cuando cambien los requerimientos)

- Si una clase asume más de una responsabilidad, las mismas se vuelven acopladas.

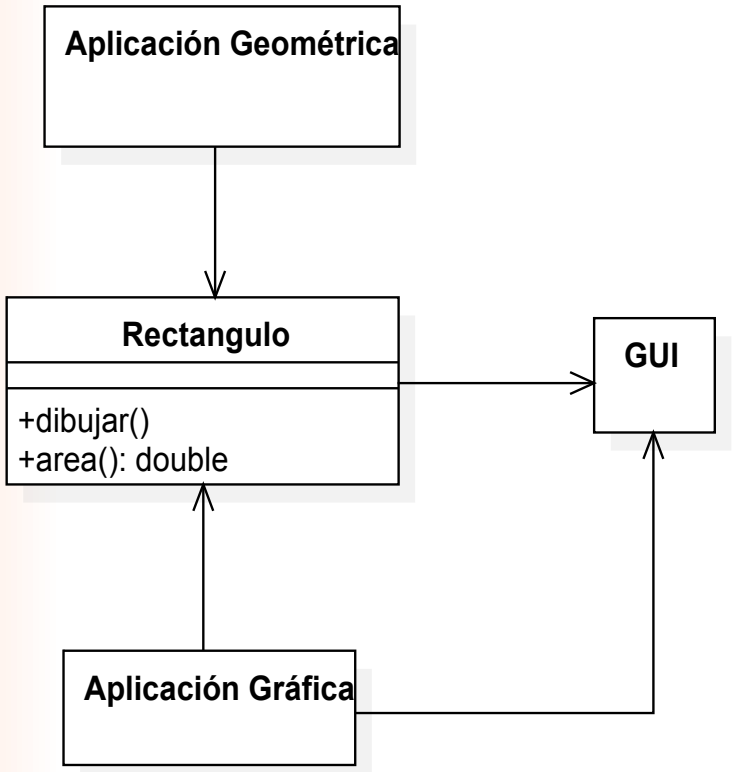
- Este principio dice que debemos huir de aquellas clases monolíticas que aglutinen varias responsabilidades.
- Si encontramos que hay más de una **razón por la que una clase pueda cambiar,** entonces es que esa clase tiene más de una responsabilidad.

Principios de Diseño SOLID

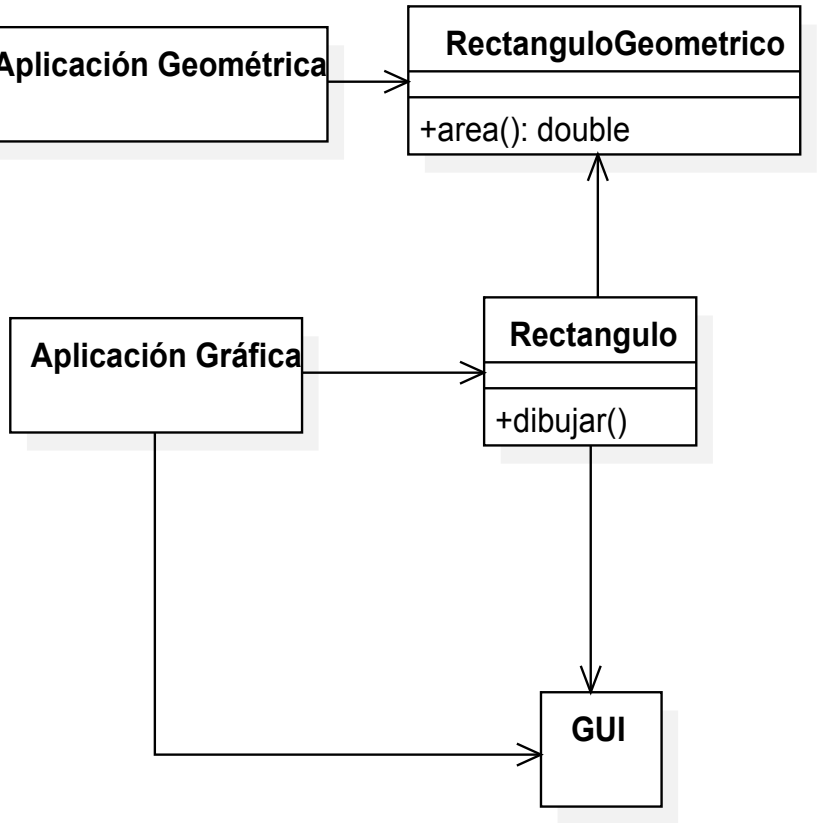
Principio de responsabilidad única

- Ejemplo Separando en 2 la Clase Rectángulo que tiene 2 responsabilidades

Más de una responsabilidad



Responsabilidades separadas



UNTDF
Universidad Nacional de Tierra del Fuego,
Antártida e Islas del Atlántico Sur

Principios de Diseño SOLID

Principio de responsabilidad única

- Ejemplo Modem

```
public interface modem
{
  public void dial (String pno)
  public void hangup ()
  public void send (char c)
  public char receive ()
}
```

} 1. responsibility (connection)

} 2. responsibility (data transfer)

```
public interface Connection
{
public void dial (String pno)
public void hangup ()
}
```

```
public interface DataChannel
{
public void send (char c)
public char receive ()
}
```

UNTDF
Universidad Nacional de Tierra del Fuego, Antártida e Islas del Atlántico Sur

Principios de Diseño SOLID

Principio Open/Closed

Las entidades software deben estar abiertas para su extensión, pero cerradas para su modificación

- Debe ser posible ***extender una entidad*** para ampliar su conjunto de operaciones y añadir nuevos atributos a sus estructuras de datos.
- Una *entidad* ***debe estar cerrada*** de forma de poder manejar cambios sin modificar su estructura central

Principios de Diseño SOLID

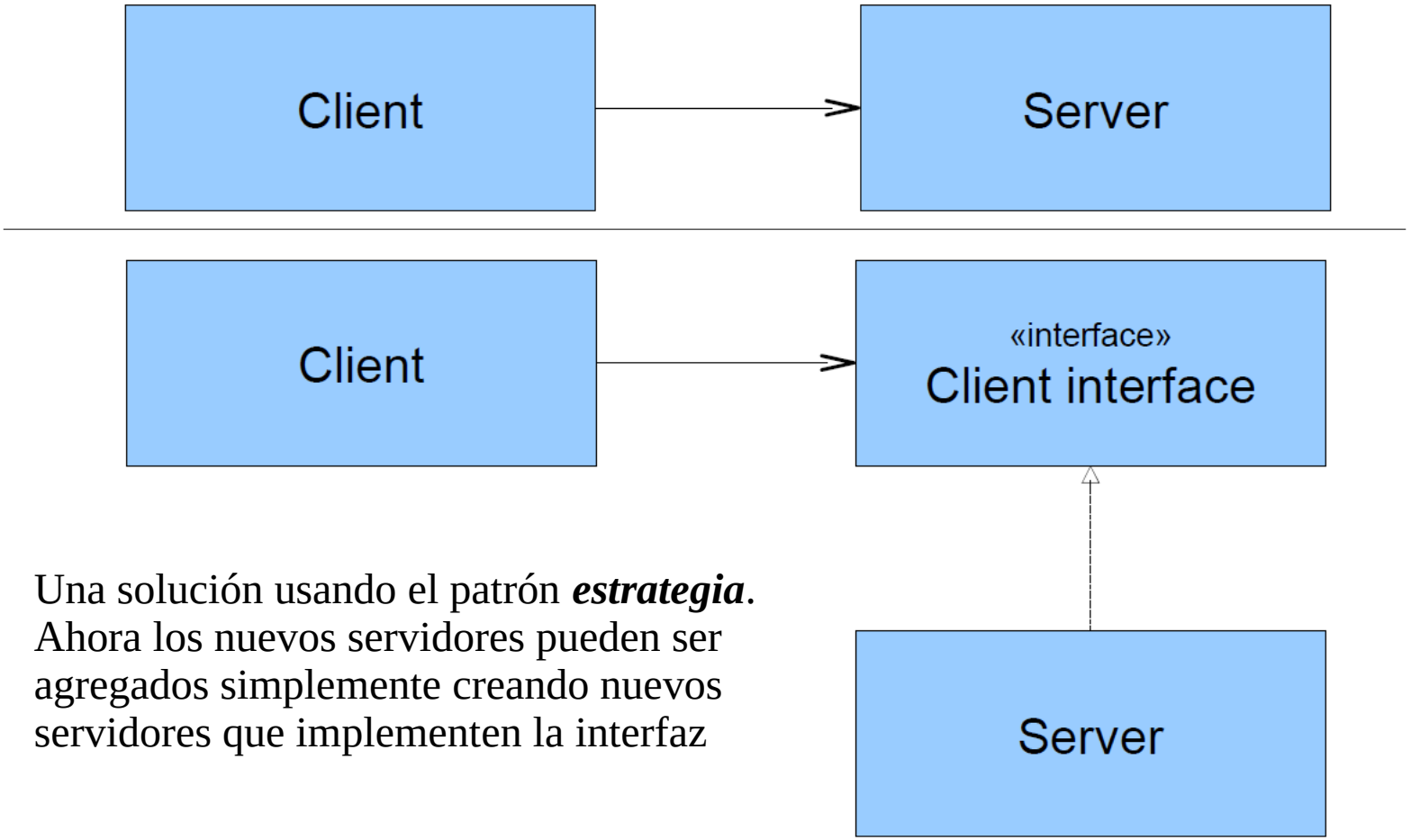
Principio Open/Closed

- Una clase debe tener una interfaz estable y bien definida (que no se modificará para no afectar a otros módulos que dependen de él)
- La clave para conseguir esto es la abstracción
    - Usando abstracciones (por ej. Interfaces en lugar de clases concretas), su estructura central puede ser extendida mediante derivaciones de dichas abstracciones.

Principios de Diseño SOLID

Principio Open/Closed

- Ejemplo: El Cliente debe cambiar cada vez que se usa un nuevo tipo de Server



Una solución usando el patrón ***estrategia***. Ahora los nuevos servidores pueden ser agregados simplemente creando nuevos servidores que implementen la interfaz

Principios de Diseño SOLID

Principio Open/Closed. Ejemplo

- Una función logOn para un Modem dentro de una clase Cliente, debe cambiarse cada vez que se agrega un nuevo tipo de Modem.

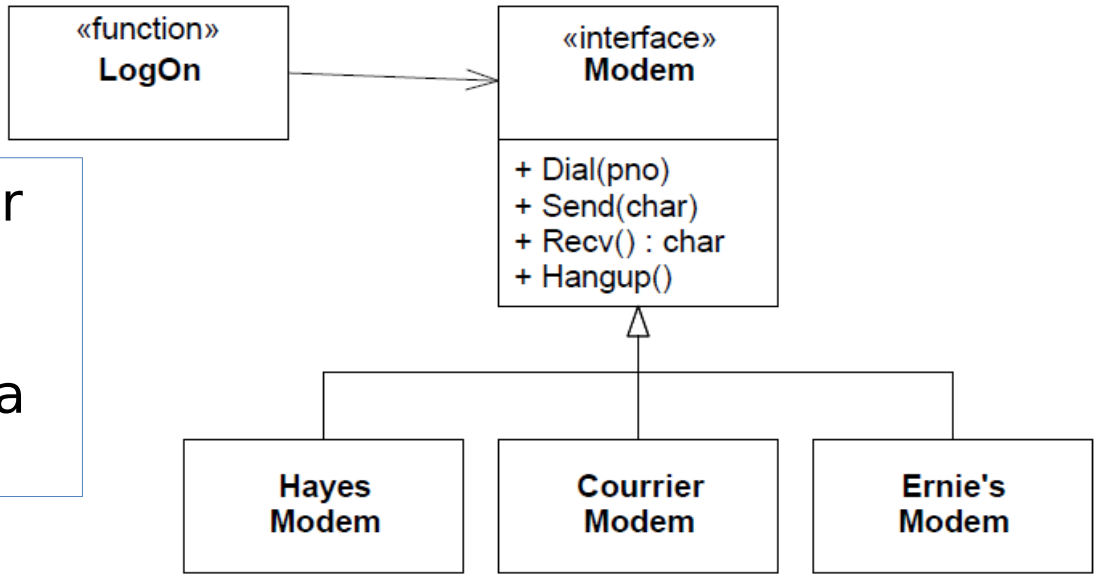
```
void LogOn(Modem& m, string& pno, string& user,
string& pw)
 {
    if (m.type == Modem::hayes)
       DialHayes((Hayes&)m,pno);
    else if (m.type == Modem::courrier)
       DialCourrier((Courrier&)m,pno);
    else if (m.type == Modem::ernie)
       DialErnie((Ernie&)m,pno);
    // ...se entiende la idea
 }
```

- Haciendo uso de la abstracción (creación de una interface Modem) y del polimorfismo, podría cerrarse a modificaciones

Principios de Diseño SOLID

Principio Open/Closed. Ejemplo

- La función logOn depende sólo de la interfaz del módulo.



Se pueden agregar otros tipos de Modem a la jerarquía sin que la función cambie

```
void LogOn(Modem& m, string& pno, string& user,
string& pw)
   {
      m.Dial(pno);
      // ...se entiende la idea
   }
```

Función LogOn cerrada a modificaciones. Se extiende agregando modems

UNTDF
Universidad Nacional de Tierra del Fuego, Antártida e Islas del Atlántico Sur

Principios de Diseño SOLID

Principio de sustitución de Liskov

Debe ser posible utilizar cualquier objeto instancia de una subclase en lugar de cualquier objeto instancia de su superclase sin que la semántica del programa escrito en los términos de la superclase se vea afectado

- Propuesto por Bárbara Liskov en el año 1987 durante una conferencia sobre Jerarquía y Abstracción de datos
- Según este principio "*Cada clase que hereda de otra puede usarse como su padre sin necesidad de conocer las diferencias entre ellas*"

Principios de Diseño SOLID

Principio de sustitución de Liskov

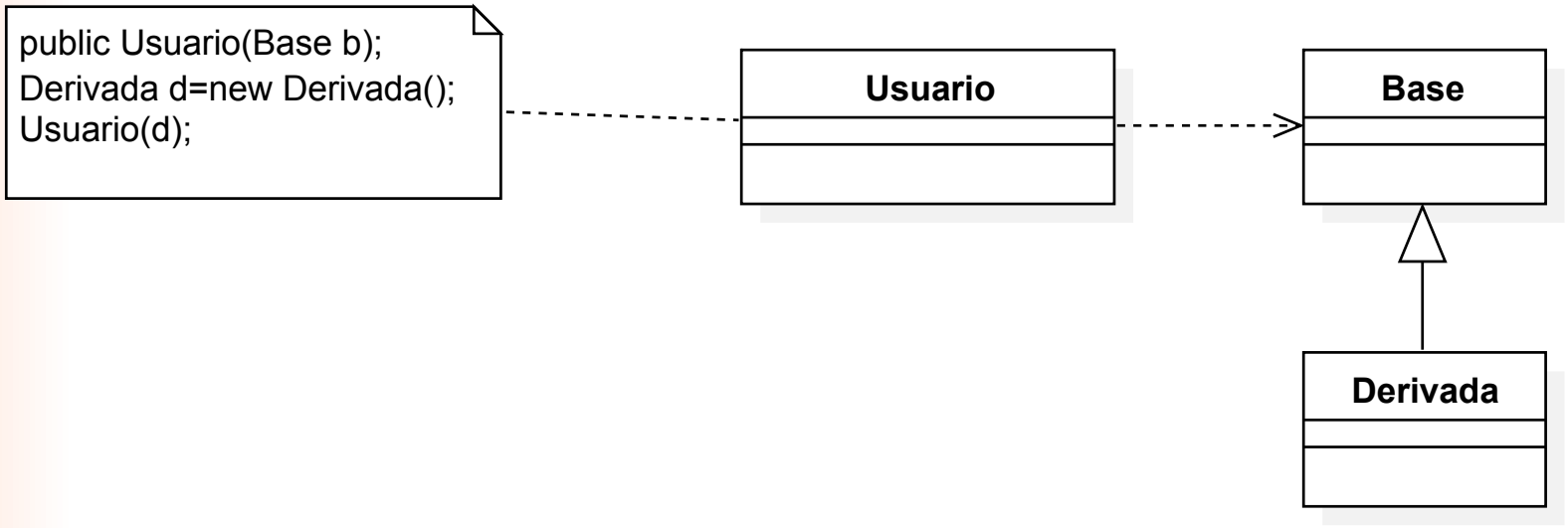
Otras definiciones del PSL

- Cada clase que hereda de otra puede usarse como su padre sin necesidad de conocer las diferencias entre ellas
- Toda subclase debe soportar ser sustituida por su clase base.
- Las funciones que usan punteros o referencias a clases base deben ser capaces de utilizar objetos de las clases derivadas sin tener conocimiento de ello" (Martin96b), también:
- Las clases derivadas deben ser utilizables a través de la interfaz de la clase base, sin necesidad de que el usuario conozca la diferencia

Principios de Diseño SOLID

Principio de sustitución de Liskov

- El Usuario de la clase base debe continuar funcionando apropiadamente si se le pasa un derivado



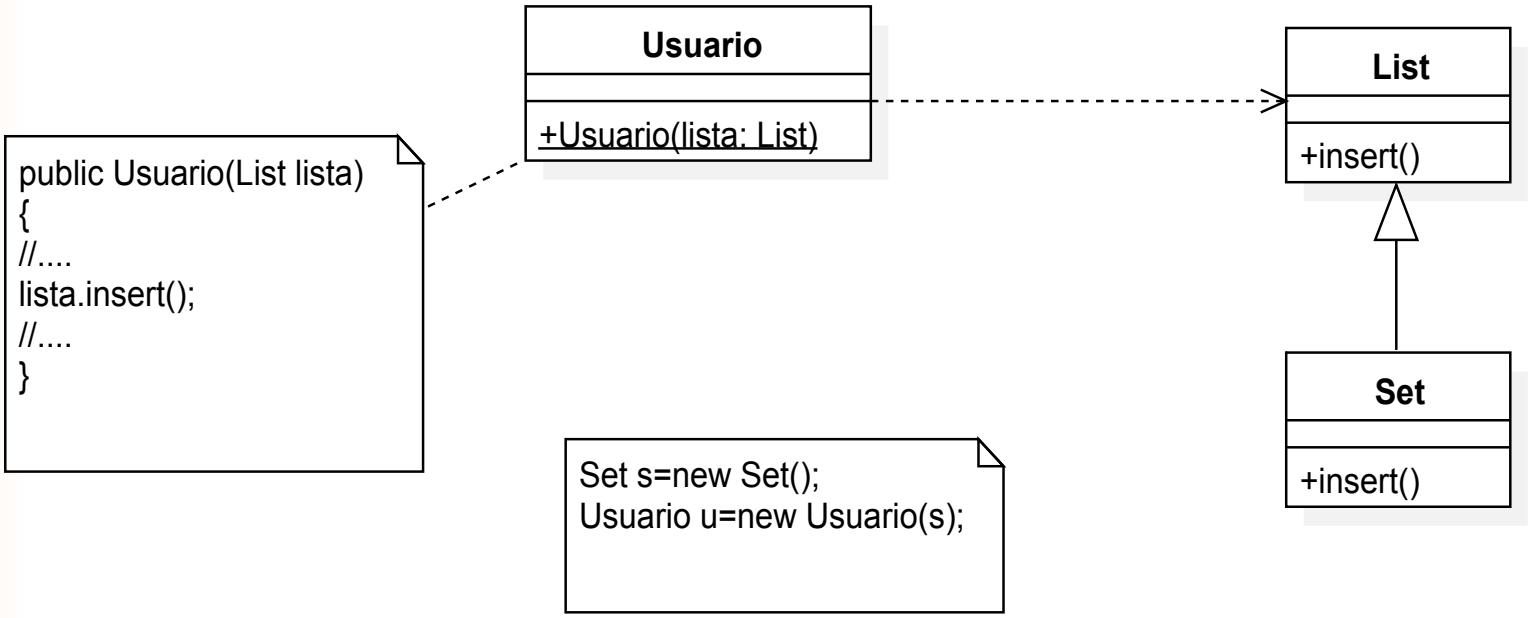
Las clases derivadas deben poder sustituirse por su clase base

UNTDF
Universidad Nacional de Tierra del Fuego, Antártida e Islas del Atlántico Sur

Principios de Diseño SOLID

Principio de sustitución de Liskov

- El Usuario de la clase base debe continuar funcionando apropiadamente si se le pasa un derivado



Se deben diseñar las clases de forma que las dependencias del cliente puedan ser sustituidas con subclases sin que el cliente se entere del cambio

Principios de Diseño SOLID

Principio de sustitución de Liskov

- Todas las subclases deben operar de la misma manera que la clase base.
- La funcionalidad específica puede ser diferente, pero por lo menos debe conformar al comportamiento esperado de la clase base.
- Para ser un verdadero subtipo de comportamiento la subclase debe no sólo implementar los métodos y propiedades de la clase base, sino conformar su comportamiento implícito.

UNTDF
Universidad Nacional de Tierra del Fuego, Antártida e Islas del Atlántico Sur

Principios de Diseño SOLID

Principio de Separación de la Interfaz

Los clientes no deben ser forzados a depender de interfaces que no utilizan
Es preferible muchas interfaces específicas de cliente que una interfaz de propósito general

- Este principio expresa que las clases que implementen una interfaz o una clase abstracta no deberían estar obligadas a tener partes que no van a utilizar.
- Una interfaz es un contrato que debe cumplir una clase, y tales contratos deben ser específicos, no genéricos; esto nos proporcionará una forma más ágil de definir una única responsabilidad por interfaz -de otra forma, violaríamos además el Principio de Responsabilidad Única

UNTDF
Universidad Nacional de Tierra del Fuego,
Antártida e Islas del Atlántico Sur

Principios de Diseño SOLID

Principio de Separación de la Interfaz

- Las clases con interfaces pesadas son aquellas cuyas interfaces no son cohesivas.
- Las interfaces pueden separarse en grupos de métodos, donde c/u sirva a un conjunto diferente de clientes
- Para ello se debería clasificar y organizar los clientes por **tipos**, creando interfaces para cada uno de estos tipos

Principios de Diseño SOLID

Principio de Separación de la Interfaz Ejemplo

```
//principio de segregacion de intefaz - mal ejemplo
interface ITrabajador{
public void trabajar();
public void comer();
}
class Trabajador implements ITrabajador{
public void trabajar(){
//....trabaja
}
public void comer(){
//......comer en descanso
}
}
class SuperTrabajador implements ITrabajador{
public void trabajar(){
//....trabaja mucho mas
}
public void comer(){
//......comer en descanso
}
}
class Manager{
ITrabajador worker;
public void setWorker(ITrabajador w){
  worker=w;
}
public void manage(){
  worker.trabajar();
}
}
```

Una nueva clase Robot, quedaría cargada con el comportamiento comer, que no utiliza

Principios de Diseño SOLID

Principio de Separación de la Interfaz Ejemplo

```
//principio de segregacion de intefaz - buen ejemplo
interface ITrabajador {
public void trabajar();
}
interface IAlimentable{
public void comer();
}
interface ITrabajadorAlim extends IAlimentable, ITrabajador {}
class Trabajador implements ITrabajador, IAlimentable{
public void trabajar(){
//....trabaja
}
public void comer(){ //......comer en el descanso
 }
}
class Robot implements ITrabajador{
public void trabajar(){//....trabaja
}
}
class SuperWorker implements ITrabajadorAlim{ //IAlimentable, ITrabajador
public void trabajar(){
//....trabaja mucho mas
}
public void comer(){
//......comer en el descanso
}
}
class Manager{
ITrabajador worker;
public void setTrabajador(ITrabajador w){
 worker=w;
}
public void manage(){
 worker.trabajar();
}
}
```

Dividiendo la Interfaz ITrabajador, Robot no carga con comportamiento que no utiliza

UNTDF
Universidad Nacional de Tierra del Fuego, Antártida e Islas del Atlántico Sur

Principios de Diseño SOLID

Principio de Inversión de Dependencia

Los módulos de alto nivel no deben depender de los módulos de bajo nivel. Ambos deben depender de las abstracciones

Las abstracciones no deben depender de los detalles. Los detalles deben depender de las abstracciones

- Este principio ayuda a mantener nuestro código totalmente desacoplado, asegurándonos que dependemos de abstracciones en vez de implementaciones concretas.

UNTDF
Universidad Nacional de Tierra del Fuego,
Antártida e Islas del Atlántico Sur

Principios de Diseño SOLID

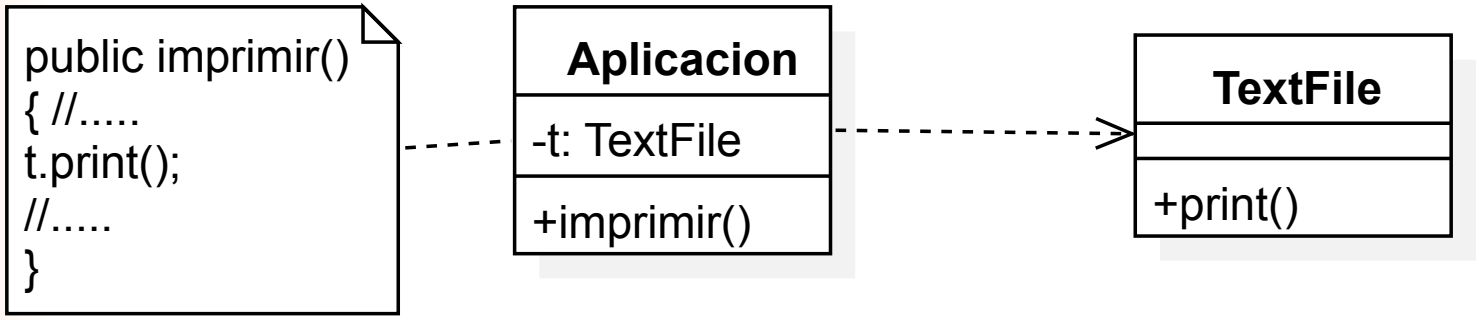
Principio de Inversión de Dependencia

- Una interpretación del principio es la estrategia de utilizar dependencias a interfaces o clases y métodos abstractos, en lugar de a clases y funciones concretas
- Según esta heurística:
  - ✔ Ninguna variable debe mantener un puntero a una referencia o clase concreta
  - ✔Ninguna clase debe derivarse de una clase concreta
  - ✔Ningún método debe sobrescribir un método implementado en la clase base
- Una motivación del principio es la dependencia de clases volátiles (como se supone a las concretas)

Principios de Diseño SOLID

Principio de Inversión de Dependencia. Ejemplo

- En el ejemplo, la Aplicación refiere a la clase concreta TextFile. Viola el PID.

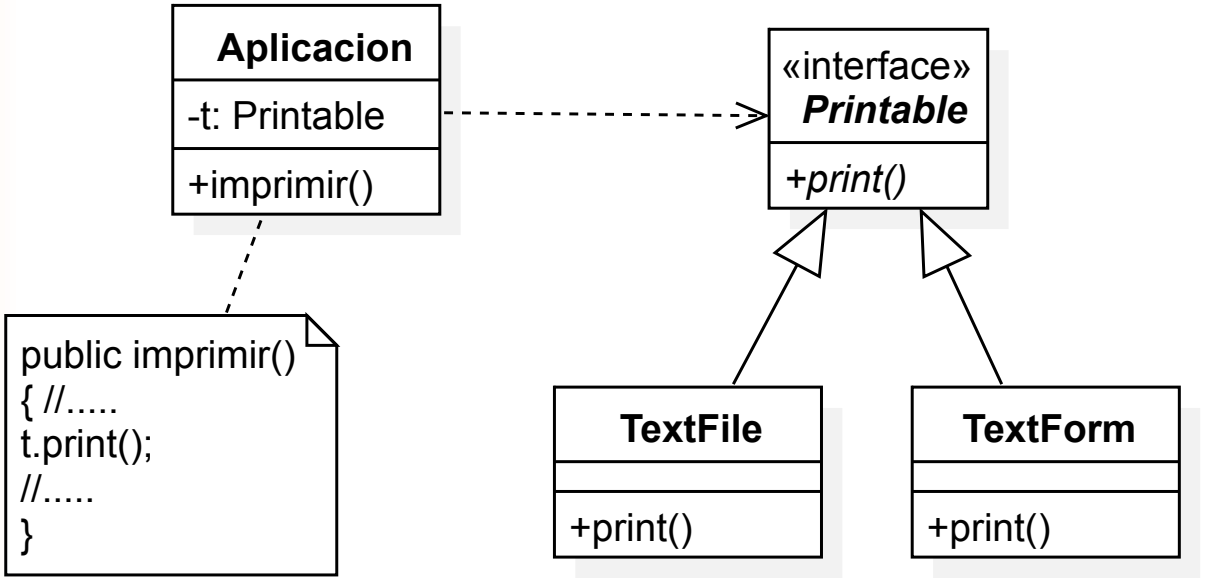


- Esta aplicación sólo podría imprimir TextFiles. Si la quisiera modificar para que también imprima TaxForms u otras cosas, debería modificarla.
- La solución es usar la abstracción para invertir la dependencia de la clase TextFile

Principios de Diseño SOLID

Principio de Inversión de Dependencia. Ejemplo

- Se incluye una Interfaz para todo lo que quiera imprimirse.



- De este modo la aplicación podrá imprimir cualquier tipo de objetos, aún de clases que aún no se crearon