PRINCIPIOS DE DISEÑO SOLID

ELSA ESTEVEZ
UNIVERSIDAD NACIONAL DEL SUR
DEPARTAMENTO DE CIENCIAS E INGENIERIA DE LA COMPUTACION

CONTENIDO

CUESTIONES PRACTICAS – 1

1) Rigidez
2) Fragilidad
3) Inmobilidad
4) Viscosidad
5) Complejidad innecesaria
6) Repeticion innecesaria
7) Opacidad

CUESTIONES PRACTICAS – 2

RIGIDEZ

- es la tendencia del software a ser difícil de cambiar, aun ante cambios simples

Ejemplo: Un único cambio causa una cascada de cambios subsecuentes en módulos dependientes. Cuantos más módulos se deben cambiar, más rígido es el diseño.

FRAGILIDAD

- es la tendencia de un sistema a romperse en varios lugares cuando se hace un cambio

Nota: Frecuentemente los nuevos problemas surgen en áreas que no tienen relación conceptual con el área que fue cambiada.

INMOBILIDAD

- un diseño es inmóvil cuando contiene partes que podrían ser útiles en otros sistemas, pero el esfuerzo y riesgo de separarlas del sistema original es muy grande.

CUESTIONES PRACTICAS – 3

VISCOSIDAD

- un software viscoso es uno en el que su diseño es difícil de preservar

Existen dos formas:

viscosidad del software - la viscosidad del diseño de un software es alta si la forma de introducir un cambio, hace que sea mas difícil preservar el diseño que vulnerarlo

viscosidad del ambiente - cuando el ambiente de desarrollo es lento e ineficiente

COMPLEJIDAD INNECESARIA

- ocurre cuando el diseño contiene elementos que no son actualmente útiles

Ejemplo: Cuando se anticipan cambios a los requerimientos y se construyen “facilidades” para manejar dichos cambios potenciales.

- en principio, parece algo bueno que previene pesadillas en futuros cambios
- muchas veces, el efecto es opuesto ya que el diseño se satura de mecanismos que nunca se usan y que el software debe mantener

CUESTIONES PRACTICAS – 4

REPETICION INNECESARIA

- cortar y pegar puede ser útil para operaciones de edición de texto, pero puede ser desastroso para operaciones de edición de código
- cuando el mismo código aparece una y otra vez, en ligeramente distintas formas, se está necesitando una abstracción
- cuando hay código redundante en el sistema, los cambios en el sistema pueden ser arduos

OPACIDAD

- es la tendencia de un módulo a ser difícil de entender
- el código puede ser escrito de una manera clara y expresiva, o de una manera compleja y opaca
- a medida que el código evoluciona en el tiempo, llega a ser más y más opaco

CONTENIDO

PRINCIPIOS SOLID

Surgieron a comienzos del año 2000 y su autor-mentor es Robert C. Martin.

El término es un acrónimo que surge de los siguientes conceptos:

S	Single Responsibility Principle	Un objeto debería tener una única responsabilidad
O	Open/Closed Principle	Las entidades de software deberían estar abiertas para extensión pero cerradas para modificación
L	Liskov Substitution Principle	Un objeto en un programa podría ser reemplazado con instancias de sus subtipos sin alterar la correctitud del programa
I	Interface Segregation Principle	Muchas interfaces específicas son mejores que interfaces de propósitos generales
D	Dependency Inversion Principle	Deberíamos depender de las abstracciones y no de las concretizaciones.

PRINCIPIOS SOLID – RELEVANCIA

- Se los considera los cinco principios básicos en el diseño y la programación orientada a objetos.

- La intención es aplicar estos principios en conjunto para que sea más probable obtener un software fácil de mantener y extender en el tiempo.

PRINCIPIOS SOLID – REGLAS BASICAS

- Los principios SOLID son guías, no son reglas inamovibles.
- Pensar y hacer cosas que tengan sentido.
- Preguntar ¿por qué...?
  - ¿Por qué hago lo que hago?
  - ¿Por qué tomé tal decisión?
  - etc.

S – SINGLE RESPONSIBILITY PRINCIPLE – 1

S – SINGLE RESPONSIBILITY PRINCIPLE – 3

DESCRIPCION – Nunca debería haber mas de una razón para que una clase cambie.

- este principio se basa en el principio de cohesión de Tom DeMarco
- si una clase tiene más de una responsabilidad, entonces las mismas quedan acopladas
- los cambios en una responsabilidad pueden afectar o inhibir la capacidad de la clase para cumplir con el resto
- una clase con alto acoplamiento conduce a diseños frágiles que se rompen de maneras inesperadas cuando se producen cambios.

Una razón de cambio es una razón sólo si el cambio ocurre.

No es prudente aplicar un principio SOLID si no hay un síntoma.

S – SINGLE RESPONSIBILITY PRINCIPLE – RESPONSABILIDAD

En el contexto de SRP, definimos una responsabilidad a una “razón de cambio”.

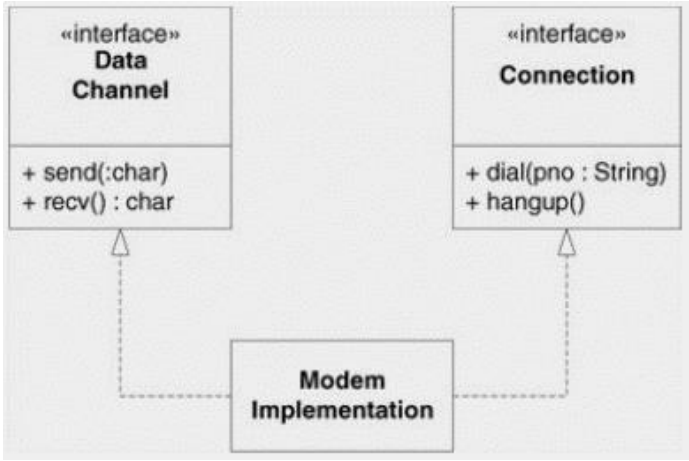
La interfaz Modem parecería se razonable:

```
Modem.java -- SRP Violation
interface Modem
{
  public void dial(String pno);
  public void hangup();
  public void send(char c);
  public char recv();
}
```

Sin embargo, existe más de una responsabilidad:

- manejo de la conexión
- comunicación de datos

S – SINGLE RESPONSIBILITY PRINCIPLE – RESPONSABILIDAD



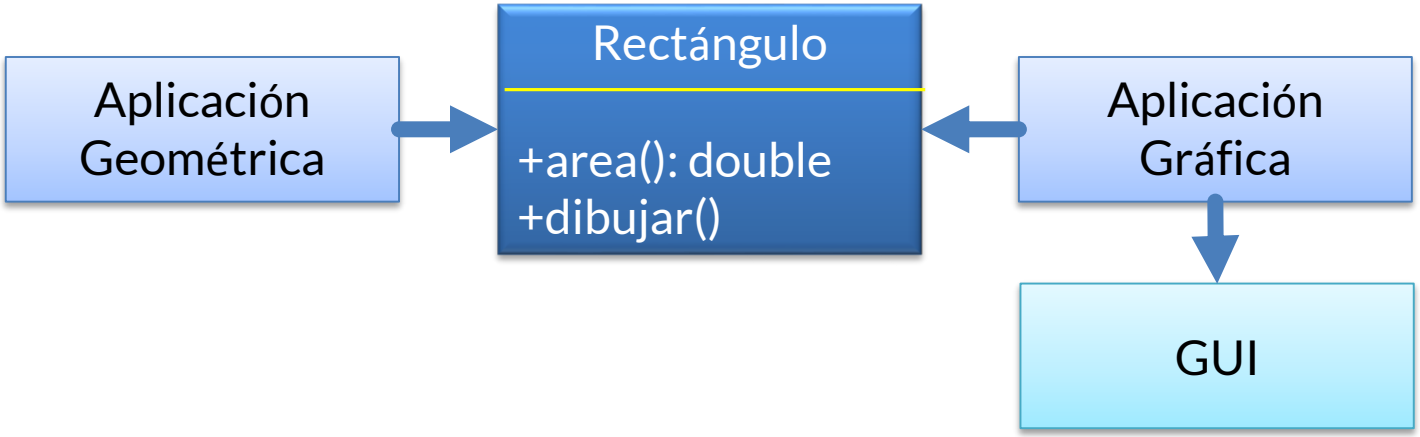
Es necesario dividir la clase?

Depende de cómo evolucione la aplicación

- Si la aplicación cambia de manera que afecta la firma de las funciones de conexión, el diseño podría parecer rígido. En este caso, sería mejor tener dos interfaces:
  - DataChannel (send, recv)
  - Connection (dial, hangup)

- Si la aplicación no cambia de manera que cause que ambas responsabilidades cambien en momentos diferentes, no habría necesidad de separarlas. Es más, separarlas podría parecer una complejidad innecesaria.

S – SINGLE RESPONSIBILITY PRINCIPLE – EJEMPLO 1



La clase Rectángulo es utilizada por dos aplicaciones diferentes:

1) Una se encarga de la geometría computacional:
   - usa Rectángulo para resolver las matemáticas de las figuras geométricas
   - nunca dibuja un rectángulo en la pantalla
2) La otra es gráfica en naturaleza. También dibuja el rectángulo en la pantalla.

Este diseño viola a SRP porque Rectángulo tiene dos responsabilidades:

i) proveer un modelo matemático para la geometría de un rectángulo

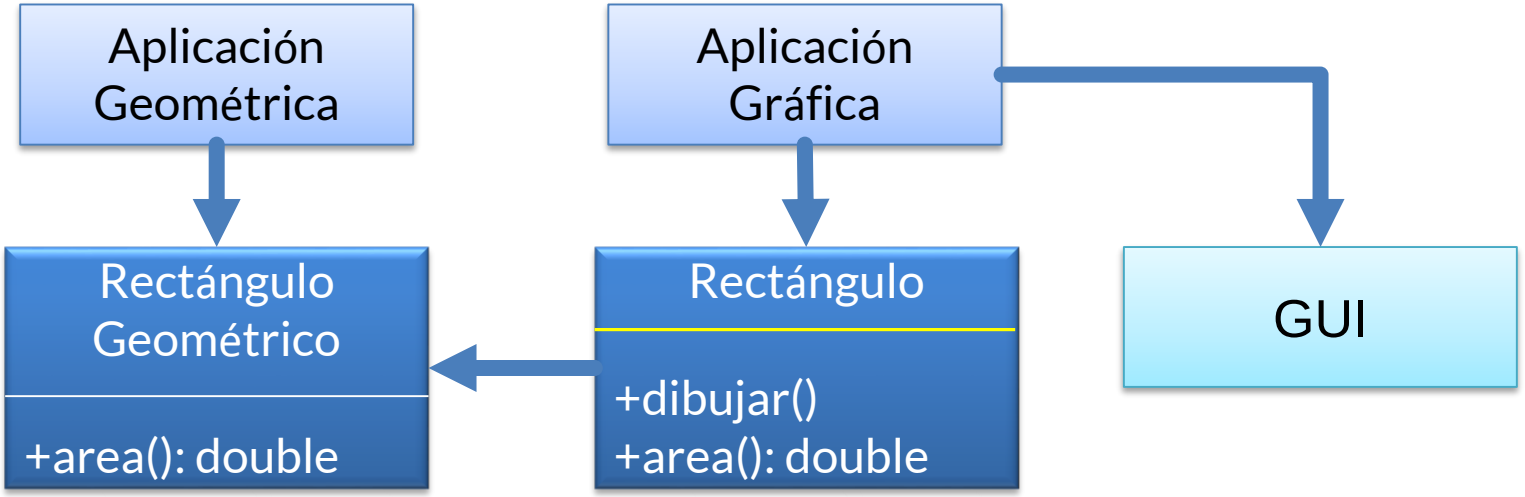
ii) dibujar el rectángulo en una interface gráfica.

S – SINGLE RESPONSIBILITY PRINCIPLE – EJEMPLO 1 SOLUCION

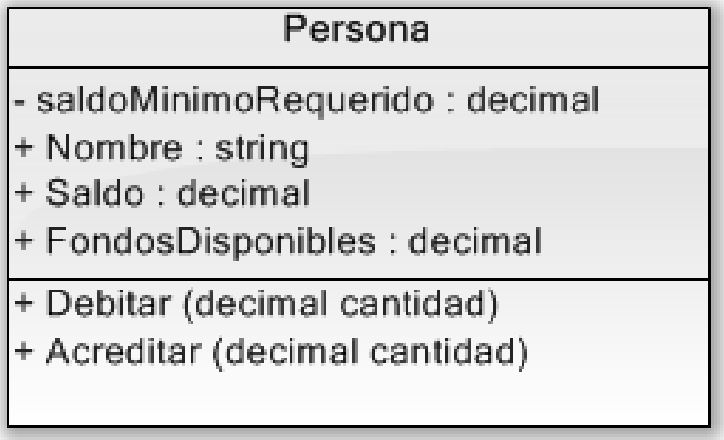
Problemas que podrían ocurrir:

- debemos incluir una dependencia con la GUI en la aplicación geométrica.
- si un cambio en la aplicación gráfica causa que cambie la clase rectángulo, podría forzarnos a re-compilar, re-testear y re-instalar la aplicación geométrica, de manera tal de asegurar su correcto comportamiento.

Posible solución: separar las responsabilidades en clases distintas



S – SINGLE RESPONSIBILITY PRINCIPLE – EJEMPLO 2

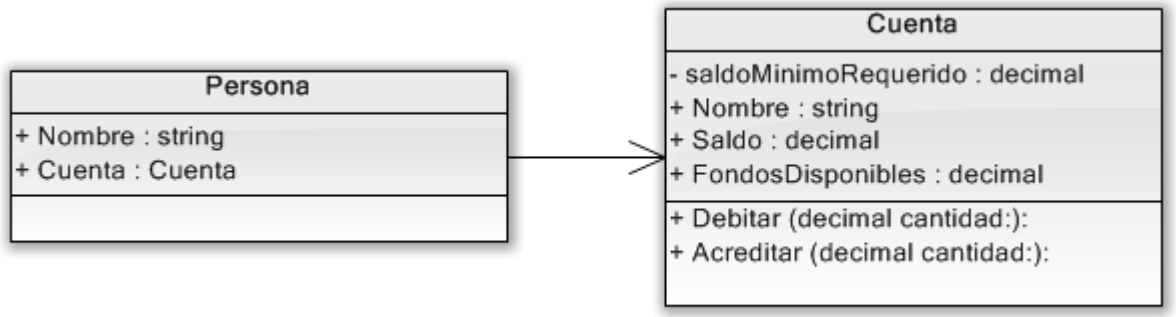


- La clase Persona maneja tanto los datos filiatorios de la persona como la información de su saldo de cuenta
- ¿Qué sucedería si ahora se permite que una cuenta sea compartida por más de una persona?

S – SINGLE RESPONSIBILITY PRINCIPLE – EJEMPLO 2 SOLUCION

- La clase Cuenta no tiene noción sobre quién la posee.

- Persona puede o bien exponer la propiedad Cuenta, o replicar la interfaz de Cuenta, delegando la implementación de sus métodos



S – SINGLE RESPONSIBILITY PRINCIPLE – EJEMPLO 3

Violación por código spaghetti

```
public class OrderProcessingModule {
  public void Process(OrderStatusMessage orderStatusMessage) {
    // Get the connection string from configuration
    string connectionString =
ConfigurationManager.ConnectionStrings["Main"].ConnectionString;

    Order order = null;
    using (SqlConnection connection = new SqlConnection(connectionString)) {
      // go get some data from the database
      order = fetchData(orderStatusMessage, connection);
    }

    // Apply the changes to the Order from the OrderStatusMessage
    updateTheOrder(order);

    // International orders have a unique set of business rules
    if (order.IsInternational)
      processInternationalOrder(order);

    // We need to treat larger orders in a special manner
    else if (order.LineItems.Count > 10)
      processLargeDomesticOrder(order);

    // Smaller domestic orders
    else
      processRegularDomesticOrder(order);

    // Ship the order if it's ready
    if (order.IsReadyToShip()) {
      ShippingGateway gateway = new ShippingGateway();
      // Transform the Order object into a Shipment
      ShipmentMessage message = createShipmentMessageForOrder(order);
      gateway.SendShipment(message);
  } }
```

S – SINGLE RESPONSIBILITY PRINCIPLE – EJEMPLO 3 SOLUCION 1

Tips para no violar el principio

- Usar capas
  - Permite lograr una primera separación de las responsabilidades de acuerdo al alcance de cada capa.
- Escribir los comentarios de código para las clases antes de comenzar a implementarlas.

```
/// <summary>
/// Gets, saves, and submits orders.
/// </summary>
public class OrderService
{
    public Order Get(int orderId) {...}
    public Order Save(Order order) {...}
    public Order SubmitOrder(Order order) {...}
}
```

- Usar métodos pequeños
  - un método debería tener un único propósito (razón para cambiar)
  - un método debería ser fácil de leer y escribir
  - escribir los pasos de un método usando verbos y sustantivos en los nombres de los métodos

S – SINGLE RESPONSIBILITY PRINCIPLE – EJEMPLO 3 SOLUCION 2

Tips para no violar el principio

- Evitar transaction scripts generalistas (que realizan muchas operaciones sobre, quizás, una misma entidad)
  - Los transaction scripts deberían tener un verbo en el nombre de la clase**.**

```
public class GetOrderService
{
    public Order Get(int orderId) { ... }
}

public class SaveOrderService
{
    public Order Save(Order order) { ... }
}

public class SubmitOrderService
{
    public Order SubmitOrder(Order order) { ... }
}
```

S – SINGLE RESPONSIBILITY PRINCIPLE

Por que es importante SRP?

- porque buscamos que sea fácil reusar código
- porque cuanto más grande es una clase, más difícil es modificarla
- porque cuánto más grande es una clase, más dura es de leer y entender.

Clases y métodos pequeños nos darán más flexibilidad, sin tener que escribir demasiado código extra.

O – OPEN CLOSED PRINCIPLE – 1

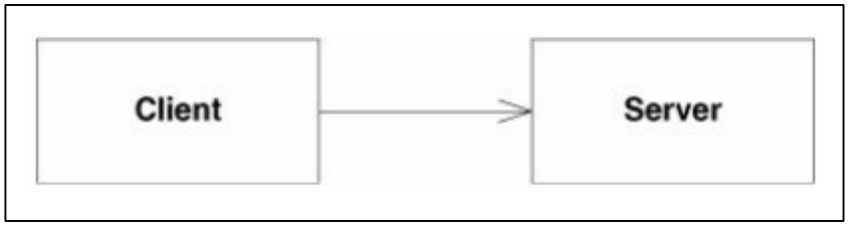
O – OPEN CLOSED PRINCIPLE – 2

DESCRIPCION – Los módulos de software debieran estar abiertos para extensión pero cerrados para modificación

- debiéramos escribir módulos que puedan ser extendidos sin necesidad de ser modificados.
- los módulos que cumplen OCP tienen dos atributos primarios:
  - están "Abiertos para Extensión" - El comportamiento del módulo puede ser extendido.
  - están "Cerrados para Modificación" - El código fuente del módulo es inviolable.

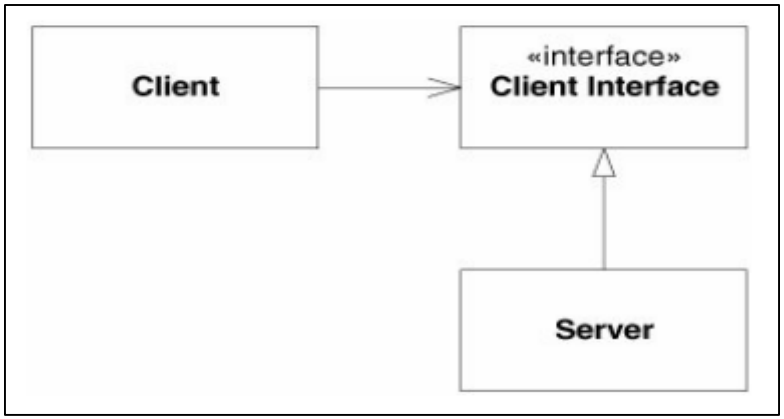
La clave es la ABSTRACCION

O – OPEN CLOSED PRINCIPLE – EJEMPLO 1



No soporta OCP

- Las clases Client y Server son clases concretas.
- Si por algún motivo la clase o implementación del Server es modificada, entonces la clase Client también debe ser modificada.

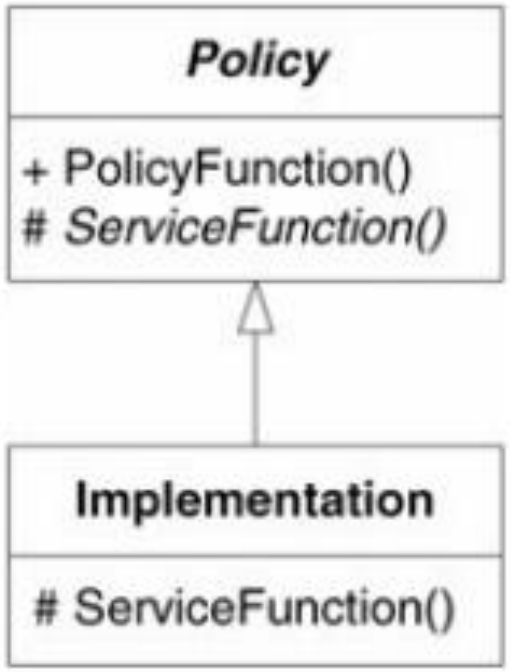


Soporta OCP

- Se agrega una interfaz intermedia, ClientInterface, entre Client y Server.
- Si, por algún motivo, la implementación del servidor cambia, el cliente probablemente no requiera cambios.
- La clase ClientInterface es cerrada para modificación aunque si está abierto para extensión.

O – OPEN CLOSED PRINCIPLE – EJEMPLO 2

- El patrón Template Method (GoF design patterns) es una alternativa clásica para lograr OCP



O – OPEN CLOSED PRINCIPLE – EJEMPLO 3

Filtros de consultas:

```
public class GetUserService
{
    public IList<UserSummary> FindUsers(UserSearchType type)
    {
        IList<User> users;
        switch (type)
        {
            case UserSearchType.AllUsers:
                // load the "users" variable here
                break;
            case UserSearchType.AllActiveUsers:
                // load the "users" variable here
                break;
            case UserSearchType.ActiveUsersThatCanEditQuotes:
                // load the "users" variable here
                break;
        }
        return ConvertToUserSummaries(users);
    }
}
```

Si se quisiera agregar un nuevo filtro, habría que modificar el enumerado y agregar el caso a la sentencia switch.

O – OPEN CLOSED PRINCIPLE – EJEMPLO 3 SOLUCION

Filtros de consultas:

```
public interface IUserFilter
{
    IQueryable<User> FilterUsers(IQueryable<User> allUsers);
}

public class GetUserService
{
    public IList<UserSummary> FindUsers(IUserFilter filter)
    {
       IQueryable<User> users = filter.FilterUsers(GetAllUsers());
       return ConvertToUserSummaries(users);
    }
}
```

Permite agregar cualquier filtro sobre la búsqueda de usuarios, sin que la consulta se entere.

O – OPEN CLOSED PRINCIPLE – EJEMPLO 4

OCP por Composición:

```
public class AuthenticationService
{
    private ILogger logger = new TextFileLogger();

    public ILogger Logger { set{ logger = value; }}

    public bool Authenticate(string userName, string password)
    {
        logger.Debug("Authentication '{0}'", userName);
        // try to authenticate the user
    }
}
public interface ILogger
{
    void Debug(string message, params object[] args);
    // other methods omitted for brevity
}
```

El servicio sólo depende de una abstracción (ILogger), sin interesarle cuál es su verdadera implementación. AuthenticationService está cerrado para modificación, pero abierto para extensión.

O – OPEN CLOSED PRINCIPLE – EJEMPLO 5

OCP por Composición con expresiones Lambda:

```
var model = new AutoPersistenceModel();
model.WithConvention(convention =>
        {
            convention.GetTableName = type => "tbl_" + type.Name;
            convention.GetPrimaryKeyName = type => type.Name +
"Id";
            convention.GetVersionColumnName = type => "Version";
        }
    );

public AutoPersistenceModel WithConvention(Action<Convention>
conventionAction)
```

- Ejemplo sacado del código de Fluent Nhibernate.
- Permite definir las convenciones de nombres tablas, claves primarias, etc.
- Sin cambiar el código de la clase AutoPersistenceModel , podemos cambiar el comportamiento del auto-mapping.
- Esta modificación del comportamiento en ejecución es posible puesto que AutoPersistenceModel depende de abstracciones (en este caso expresiones lambda), y no sobre implementaciones específicas.

O – OPEN CLOSED PRINCIPLE – EJEMPLO 5

OCP por Herencia:

```
public class Line
{
    public void Draw(ICanvas canvas) { /* draw a line on the canvas
*/ }
}

public class Painter
{
    private IEnumerable<Line> lines;

    public void DrawAll()
    {
        ICanvas canvas = GetCanvas();
        foreach (var line in lines)
        {
            line.Draw(canvas);
        }
    }
}
```

Que pasaría si se quiere agregar un rectángulo?

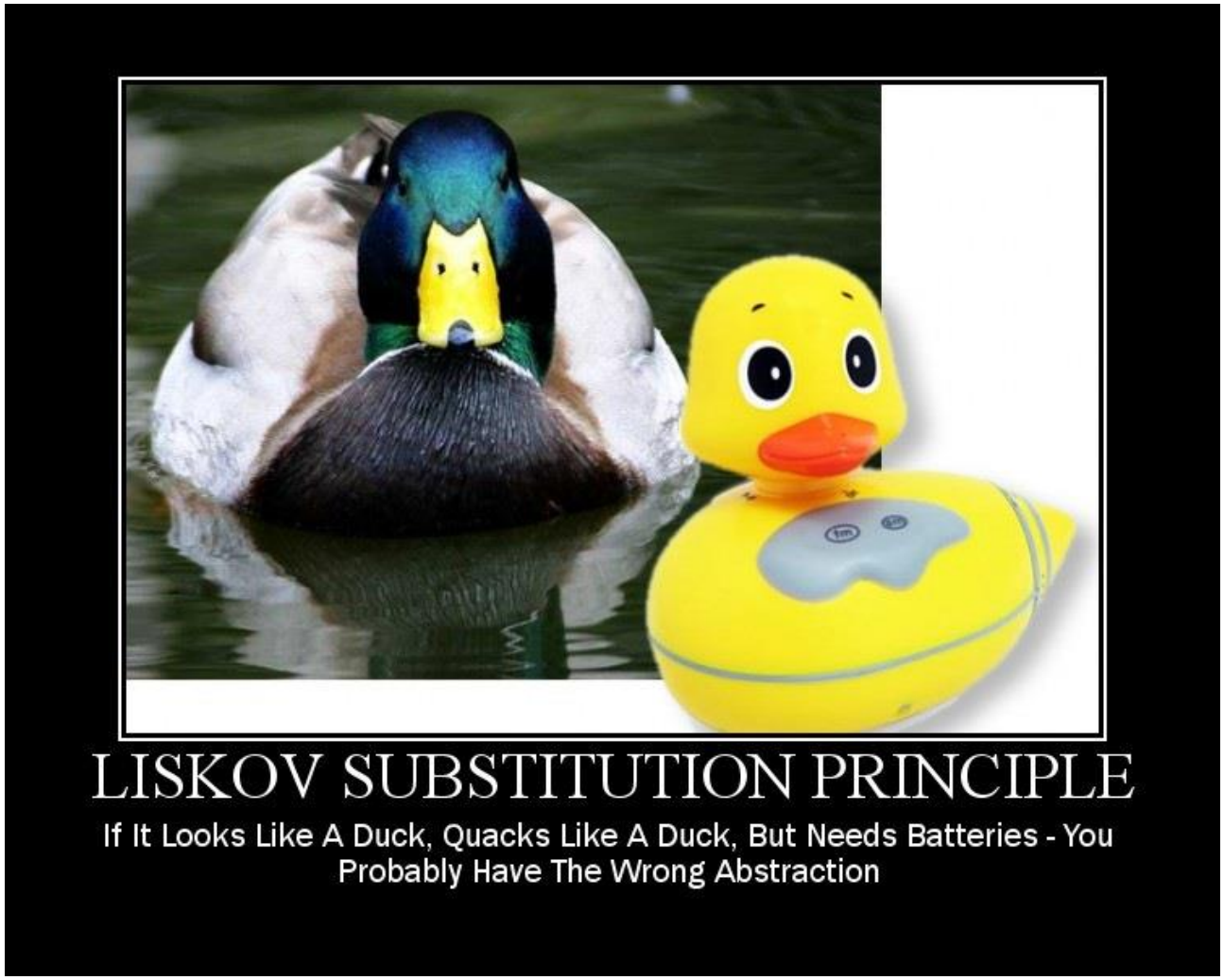
S – OPEN CLOSED PRINCIPLE

Por que es importante OCP?

- Porque la modificación de código existente y funcionando puede introducir bugs
- A veces, necesitamos modificar librerías de terceros o nuestras, pero no podemos/queremos generar una nueva versión de las mismas

Diseños extensibles son menos propensos a errores ante cambios de requerimientos.

L – LISKOV SUBSTITUTION PRINCIPLE – 1



L – LISKOV SUBSTITUTION PRINCIPLE – 2

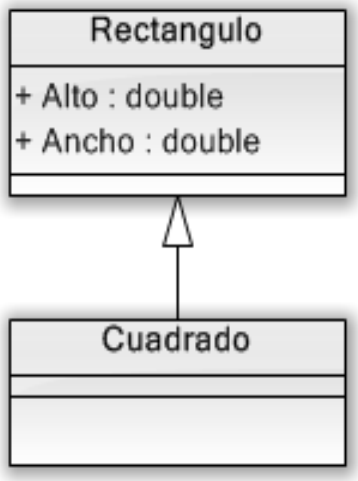
DESCRIPCION – Subclases deberían poder ser substituidas por sus clases base.

- fue acuñado por Barbara Liskov y está relacionado al concepto “Diseño por Contratos” de Bertrand Meyer

El contrato de una clase base debe ser honrado por sus clases derivadas.

L – LISKOV SUBSTITUTION PRINCIPLE – EJEMPLO 1

- la herencia generalmente se interpreta como una relación “es un”
- un Cuadrado “es un” Rectángulo
- sin embargo:
  - Cuadrado no necesita tener un Alto y un Ancho, sólo le alcanza con un Lado → desperdicio de memoria
  - a un Cuadrado se le puede setear el Alto y el Ancho →Problema de consistencia !!!!
- Solución: Sobreescribir esas propiedades en la clase Cuadrado




```
public override double Alto
{
    get { return base.Alto; }
    set
    {
        base.Alto = value;
        base.Ancho = value;
    }
}
```

```
public override double Ancho
{
    get { return base.Ancho; }
    set
    {
        base.Ancho = value;
        base.Alto = value;
    }
}
```

L – LISKOV SUBSTITUTION PRINCIPLE – EJEMPLO 1 ANALISIS

- Qué problema se puede identificar con la siguiente función cuando se le pasa una instancia de Cuadrado?

```
public void Calcular(Rectangle r)
{
    r.Alto = 5;
    r.Ancho = 4;
    Debug.Assert(r.Alto * r.Ancho == 20);
}
```

L – LISKOV SUBSTITUTION PRINCIPLE – EJEMPLO 1 ANALISIS

- ¿Podría un desarrollador asumir que cuando se cambia el valor al Alto de un Rectángulo realmente no se cambia?
- Se ha violado el contrato de Rectángulo → Violación de LSP !
- Cuidado! La validez de un modelo no es intrínseca
  - Un modelo, visto de manera aislada, no puede ser validado significativamente.
  - La validez del modelo sólo puede ser expresada en función de sus clientes.
- ¿Qué falló?
  - Un cuadrado puede ser un rectángulo, pero un objeto Cuadrado NO ES un objeto Rectángulo, ya que el comportamiento del objeto Cuadrado no es consistente con el comportamiento del objeto Rectángulo.

DISEÑO POR CONTRATO

DEFINICION

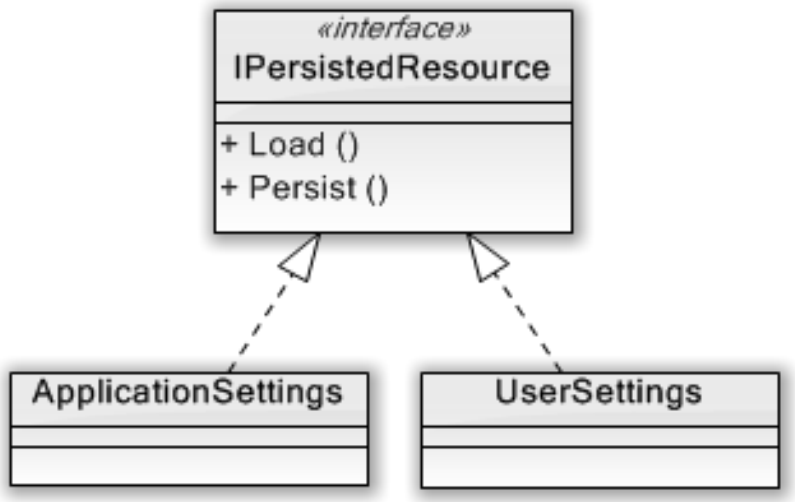
En un diseño por contrato, los métodos definirían pre-condiciones y post-condiciones.

Cuando se redefine un método [en una clase derivada], sólo se pueden reemplazar su pre-condición por una más débil y su post-condición por una más fuerte.

Si el lenguaje no permite definir las aserciones, mínimamente debieran quedar documentadas en el código.

L – LISKOV SUBSTITUTION PRINCIPLE – EJEMPLO 2

Modelo:



Cliente:

```
void SaveAll(List<IPersistedResource> res)
{
    res.ForEach(r => r.Persist());
}

List<IPersistedResource> LoadAll()
{
    var all = new List<IPersistedResource>
    {
        new UserSettings(),
        new ApplicationSettings()
    };
    all.ForEach(r => r.Load());
    return all;
}
```

L – LISKOV SUBSTITUTION PRINCIPLE – EJEMPLO 2

¿Qué pasa si quiero agregar la clase ReadOnlySettings al modelo?

```
public class ReadOnlySettings : IPersistedResource
{
    public void Load()
    {
        // stuff...
    }
    public void Persist()
    {
        throw new NotImplementedException();
    }
}
```

Debemos cambiar el cliente:

```
List<IPersistedResource> LoadAll()
{
    var all = new List<IPersistedResource>
    {
        new UserSettings(),
        new ApplicationSettings(),
        new ReadOnlySettings ()
    };
    all.ForEach(r => r.Load());
    return all;
}
```

L – LISKOV SUBSTITUTION PRINCIPLE – EJEMPLO 2

¿Qué sucede con el método SaveAll del cliente?

CANCELA con una NotImplementedException !

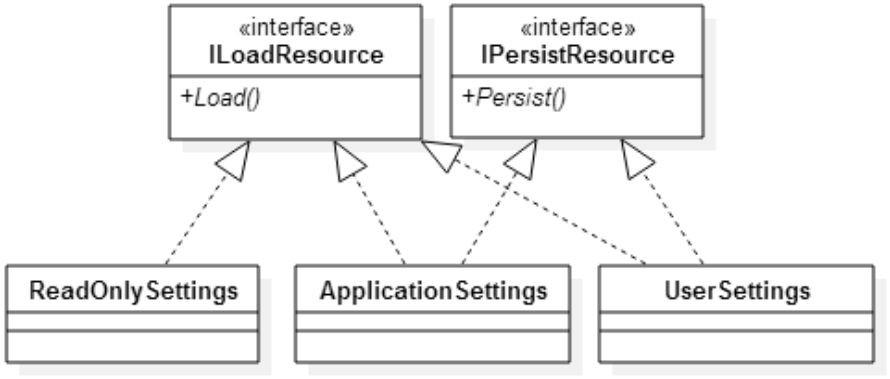
SOLUCION - Manejar el caso diferente en el cliente (método SaveAll):

```
void SaveAll(List<IPersistedResource> res)
{
    res.ForEach(r =>
                        {
                            if (r is ReadOnlySettings)
                                return;
                            r.Persist();
                        });
}
```

No es una manera muy acertada de sobrellevar el problema!

L – LISKOV SUBSTITUTION PRINCIPLE – EJEMPLO 2

MEJOR SOLUCION - Separar la interfaz IPersistedResource en IPersistResource e ILoadResource y asignarle a cada cliente la interfaz más apropiada.




```
List<ILoadResource> LoadAll()
{
    var all = new List<ILoadResource>
    {
        new UserSettings(),
        new ApplicationSettings(),
        new ReadOnlySettings ()
    };
    all.ForEach(r => r.Load());
    return all;
}
```

```
void SaveAll(List<IPersistResource> res)
{
    res.ForEach(r => r.Persist());
}
```

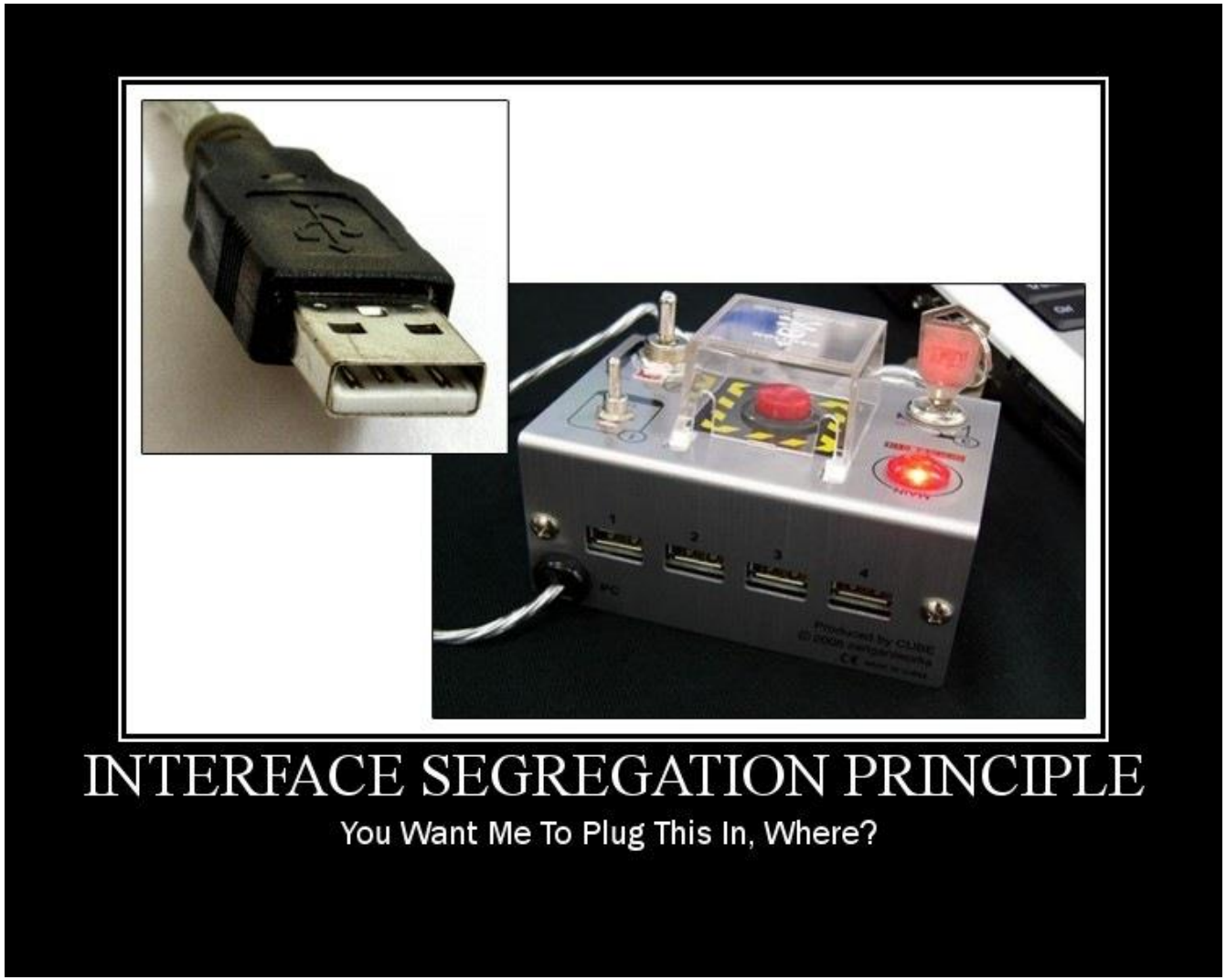
L – LISKOV SUBSTITUTION PRINCIPLE

Por que es importante LSP?

- es una característica importante de todos los programas que cumplen con OCP
- permite definir herencias consistentes

CONTENIDO

I – INTERFACE SEGREGATION PRINCIPLE – 1



I – INTERFACE SEGREGATION PRINCIPLE – 2

DESCRIPCION – Varias interfaces específicas a los clientes son mejores que una sola interface de propósito general.

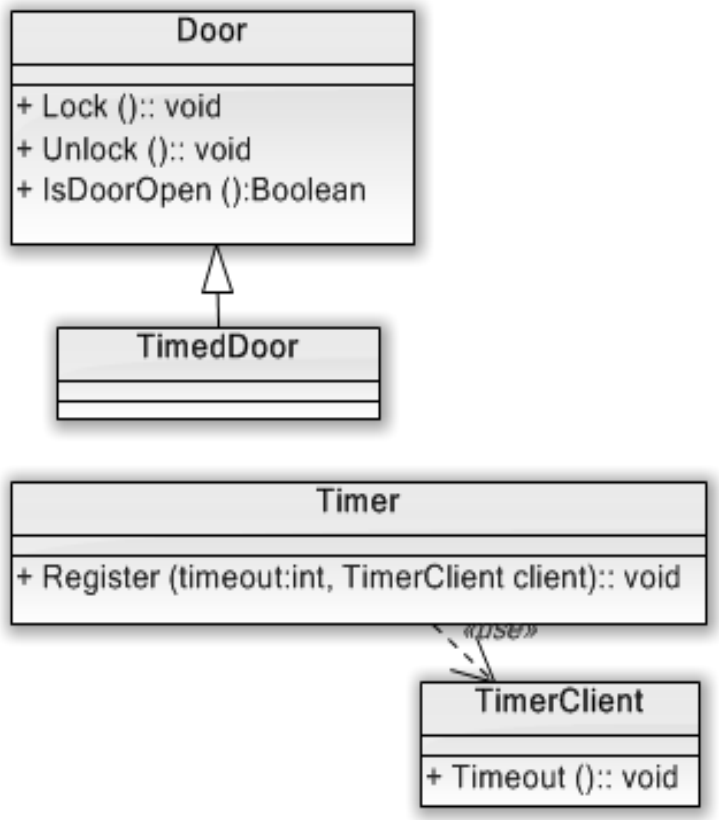
- ataca las desventajas de interfaces gordas (no cohesivas / contaminadas)
- la interface “gorda” es dividida en grupos de métodos cohesivos. Cada grupo sirve a un tipo de cliente específico.

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

I – INTERFACE SEGREGATION PRINCIPLE – EJEMPLO

Escenario de polución de interfaces

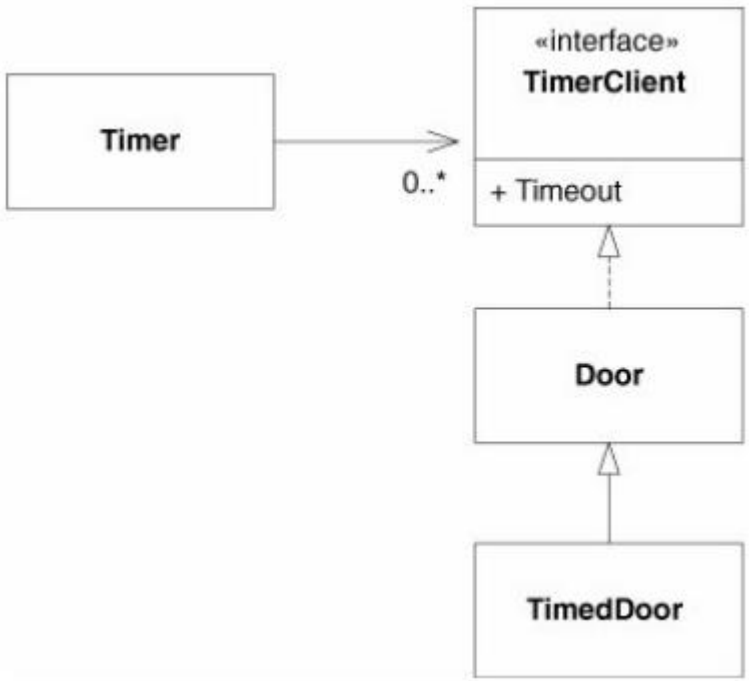
- Supongamos tener las abstracciones Door y Timer
- Se desea agregar una implementación específica de TimedDoor
- ¿Cómo implementar la relación entre TimerClient y TimedDoor?



I – INTERFACE SEGREGATION PRINCIPLE – SOLUCION 1

SOLUCION 1 – Herencia

- No todas las variedades de puerta requieren timing. La abstracción Door no tiene nada que ver con timings.
- Todas las sub-clases de Door tendrán que implementar el método Timeout.
- La interface Door ha sido contaminada por la interface TimerClient porque una de sus sub-clases lo requiere.



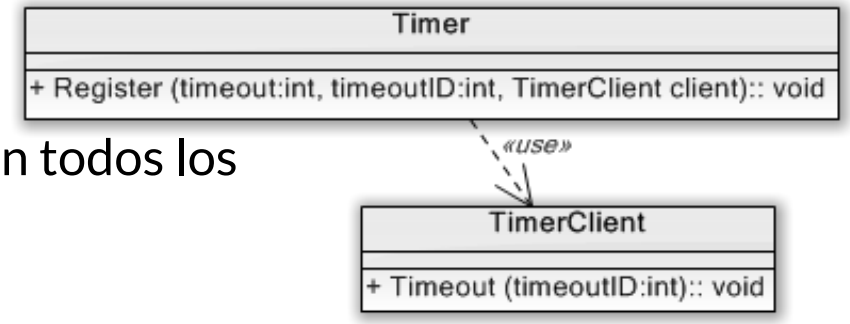
I – INTERFACE SEGREGATION PRINCIPLE – SOLUCION 1 ANALISIS

¿Qué sucede si una puerta necesita registrarse más de una vez en el timer?

- Se puede agregar un timeoutID a cada registración para saber de dónde se produjo un timeout.

- Implica modificar todas las implementaciones de TimerClient… incluso aquellas puertas que no tiene nada que ver con timings.

- Aun más, se van a ver afectados también todos los clientes de Door !!!!



Cuando un cambio en un componente afecta a otros componentes que no están relacionados, el costo de repercusión de cambios se torna impredecible, y el riesgo de romper algo se incrementa dramáticamente.

I – INTERFACE SEGREGATION PRINCIPLE – SOLUCION 2

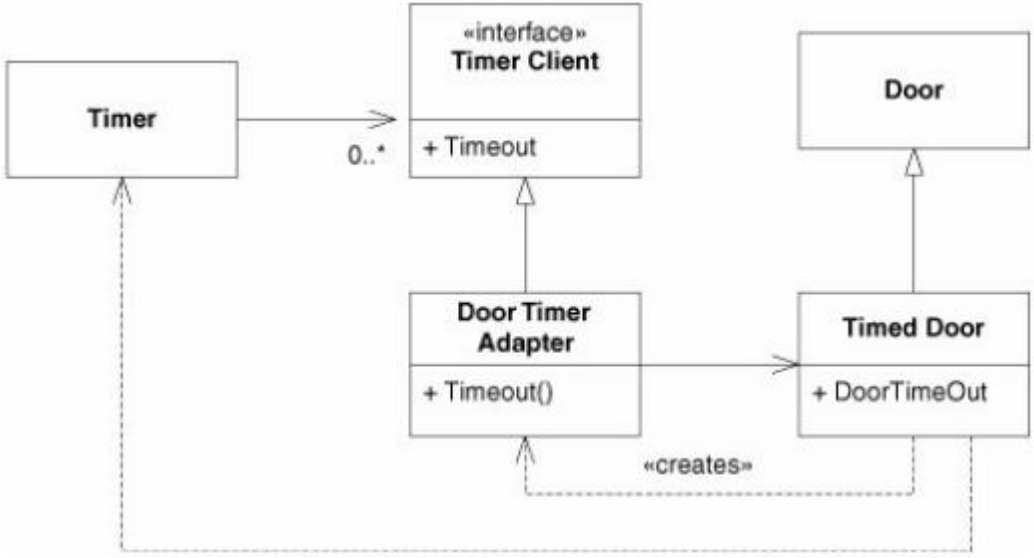
¿De qué manera se podría resolver el problema anterior?

- TimedDoor tiene dos interfaces utilizadas por dos tipos de cliente distintos: Timer y los clientes de Door.
- Los clientes de TimedDoor no necesitan acceder a las dos interfaces a la vez, sólo necesitan conocer una o la otra.
- Esto nos llevaría a separar las interfaces y que cada tipo de cliente consuma la interfaz que necesita.

I – INTERFACE SEGREGATION PRINCIPLE – IMPLEMENTACION 1

Separación a través de delegación

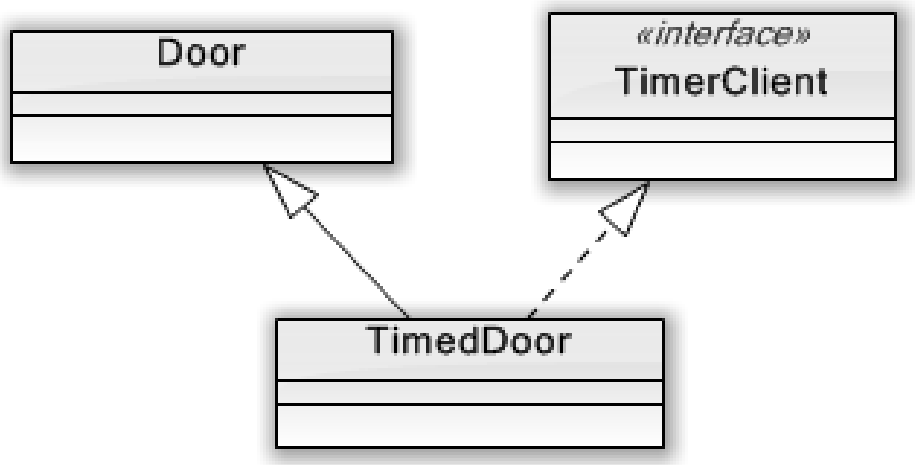
- aplica el patrón Adapter de GoF
- evita el acoplamiento de Door y Timer.
- si ocurriera un cambio en TimerClient, ningún cliente de Door se vería afectado.
- más aun, TimedDoor no tiene que tener la misma interface propuesta por TimerClient, ya que el adapter podría transformarla.



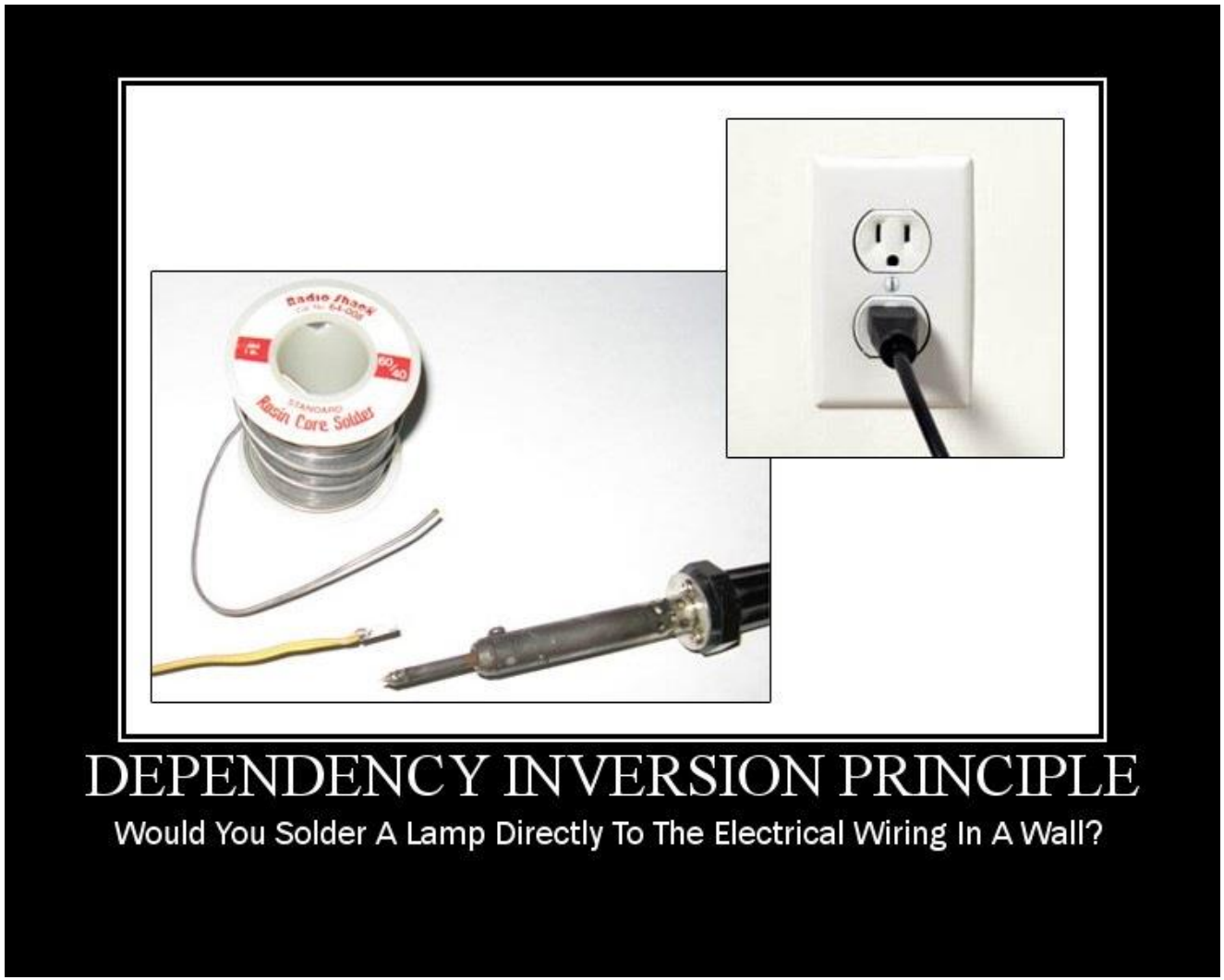
I – INTERFACE SEGREGATION PRINCIPLE – IMPLEMENTACION 2

Separación a través de Herencia Múltiple

- implementando herencia múltiple con interfaces
- desacopla las interfaces, aunque se siguen implementando ambas en el mismo objeto.
- es más elegante y mas utilizada que la solución anterior



D – DEPENDENCY INVERSION PRINCIPLE – 1



D – DEPENDENCY INVERSION PRINCIPLE – 2

DESCRIPCION – Módulos de alto nivel no debieran depender de módulos de bajo nivel. Ambos debieran depender de abstracciones.

Las abstracciones no debieran depender de los detalles, sino los detalles debieran depender de las abstracciones.

Objetivo:

- atacar el alto acoplamiento
- cada dependencia en el diseño debería apuntar a una abstracción.

Motivación:

- las clases concretas tienen mayor probabilidad de cambiar que las abstracciones.

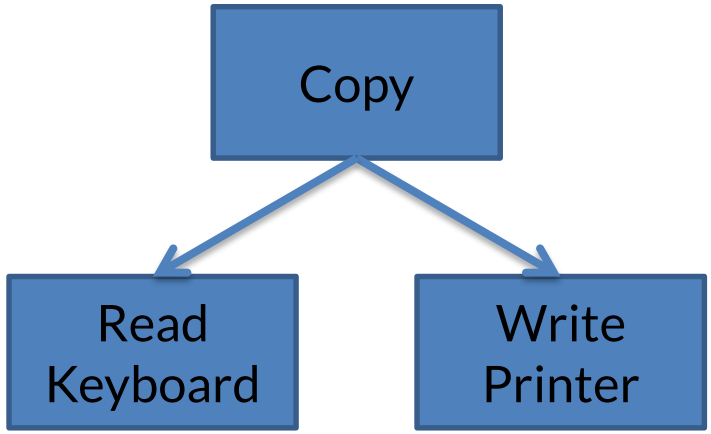
No debe aplicarse a rajatabla sobre todas las clases porque elevaría la complejidad y disminuiría la legibilidad, entre otras cosas.

D – DEPENDENCY INVERSION PRINCIPLE – EJEMPLO 1

- copiar caracteres ingresados por el teclado en la impresora

- ¿Qué pasa si se quisiera copiar los datos a un archivo de disco?
  - Copy no es reusable en contextos que no contemplen un teclado y una impresora porque depende del teclado y de la impresora.
  - Podríamos agregar una sentencia "if" en el cuerpo del ciclo
    - Poco escalable. Si se agregan más dispositivos, el "if" se hace enorme.

```
void Copy()
{
  int c;
  while ((c = ReadKeyboard()) != EOF)
    WritePrinter(c);
}
```

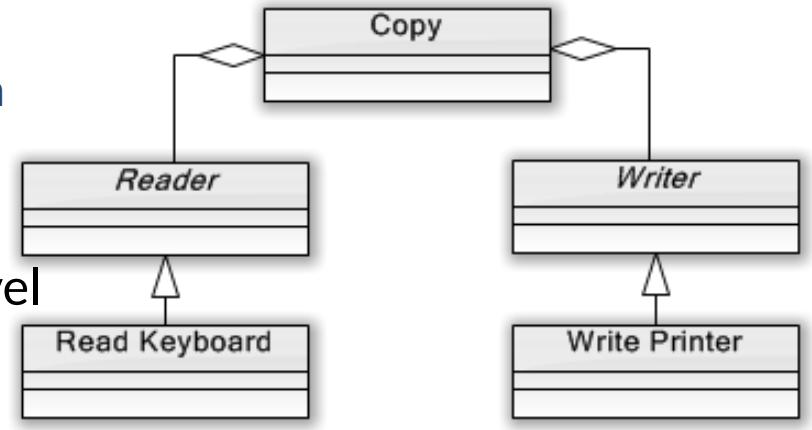


D – DEPENDENCY INVERSION PRINCIPLE – SOLUCION

¿De qué manera se podría resolver el problema anterior?

El módulo que contiene la política de alto nivel (Copy) no dependa de los detalles de implementación.

- Las dependencias se invierten:
  - Copy depende de abstracciones.
  - los detalles dependen de las mismas abstracciones
- Copy puede ser reusado en otros contextos y con otras implementaciones de Reader y Writer

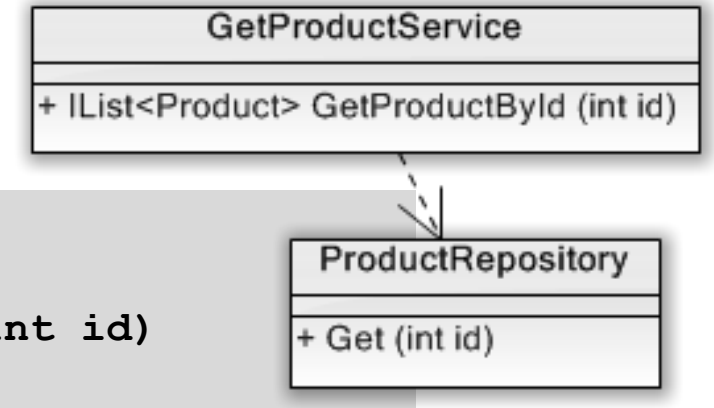



```
class Reader
{
  public:
    virtual int Read() = 0;
};

class Writer
{
  public:
    virtual void Write(char) = 0;
};

void Copy(Reader& r, Writer& w)
{
  int c;
  while((c=r.Read()) != EOF)
    w.Write(c);
}
```

D – DEPENDENCY INVERSION PRINCIPLE – EJEMPLO 2

El servicio de domino depende directamente de la implementación del repositorio.

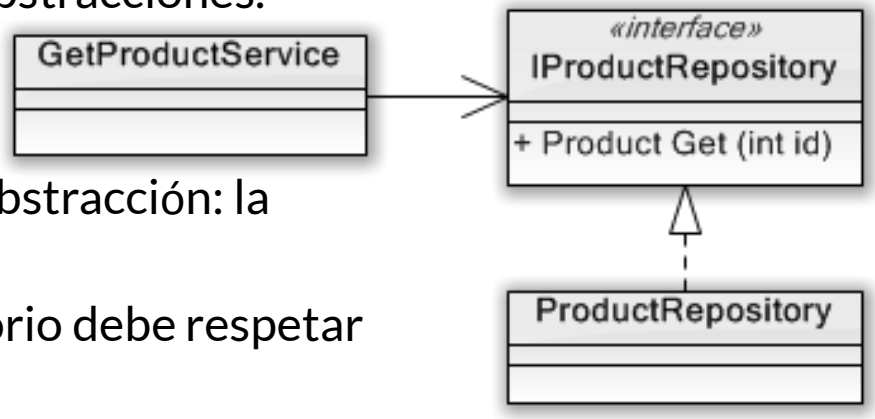
- es imposible testear unitariamente al servicio de dominio sin involucrar al repositorio
- cualquier cambio en el repositorio puede afectar directamente al servicio de dominio.
- esto viola el DIP




```
public class GetProductService
{
    public IList<Product> GetProductById(int id)
    {
        var productRepository = new ProductRepository();
        return productRepository.Get(id);
    }
}
```

D – DEPENDENCY INVERSION PRINCIPLE – SOLUCION

SOLUCION – Usar interfaces para definir abstracciones.



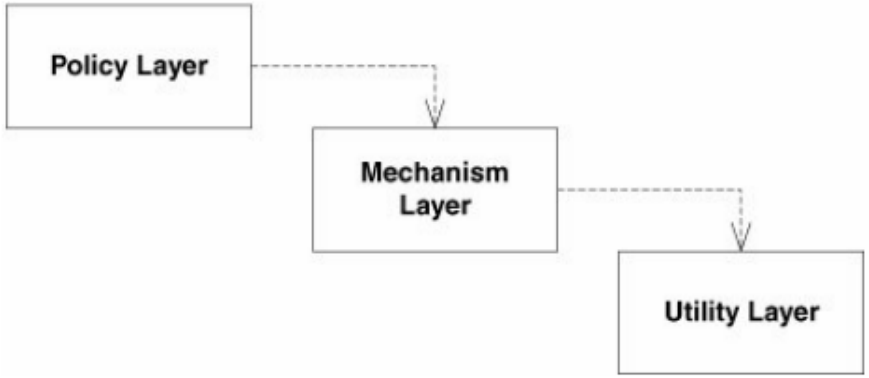
- el servicio de dominio depende de una abstracción: la interfaz del repositorio.
- la implementación concreta del repositorio debe respetar esa interface.
- la dependencia se le inyecta al servicio de dominio

```
public class GetProductService : IGetProductService
{
    private IProductRepository productRepository;
    public GetProductService(
                IProductRepository productRepository)
    {
        this.productRepository = productRepository;
    }
    public IList<Product> GetProductById(int id)
    {
        return this.productRepository.Get(id);
    }
}
```

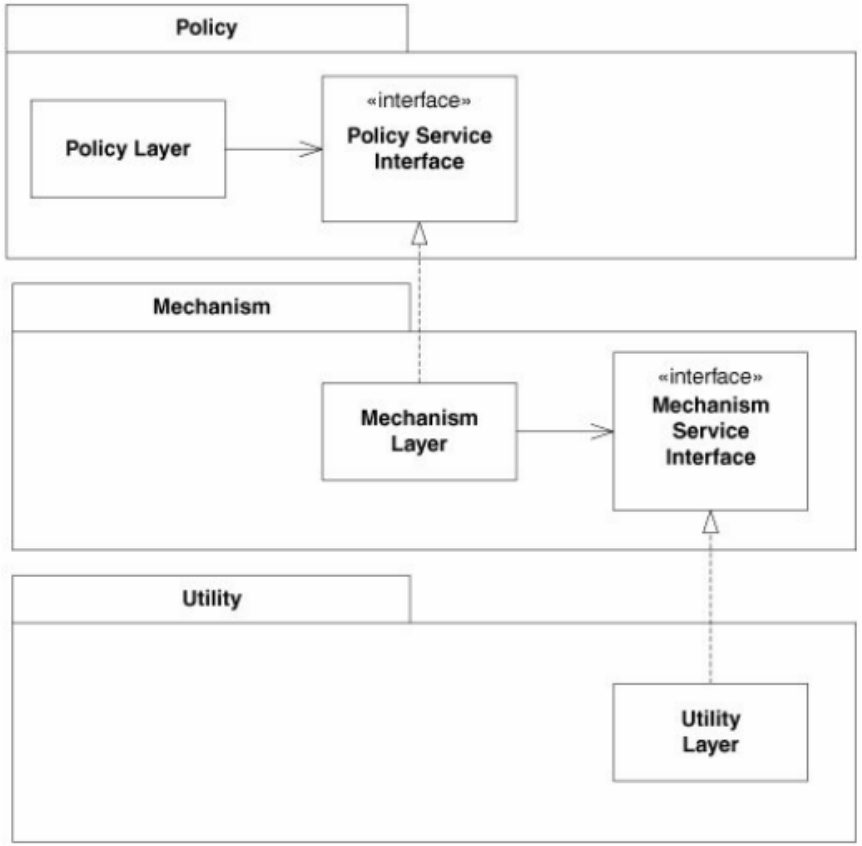
D – DEPENDENCY INVERSION PRINCIPLE – EJEMPLO 3

LAYERING

Solución simple



Solución con DIP



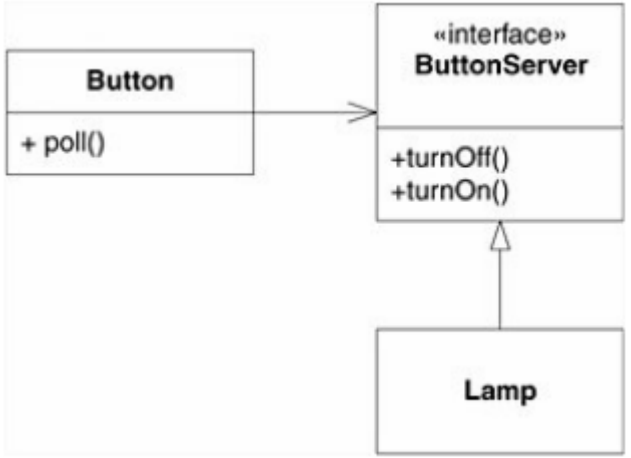
D – DEPENDENCY INVERSION PRINCIPLE – EJEMPLO 4

PROBLEMA DE LA LAMPARA - Cuando se le envía el mensaje Poll al botón, determina si el usuario lo ha presionado y enciende/apaga la lámpara.

Solución simple


```
public class Button
{
   private Lamp lamp;
   public void Poll()
   {
      if (/*some condition*/)
      lamp.TurnOn();
   }
}
```

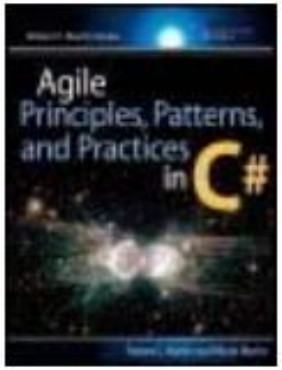
Solución con DIP



Podriamos cambiar ButtonServer por SwitchableDevice

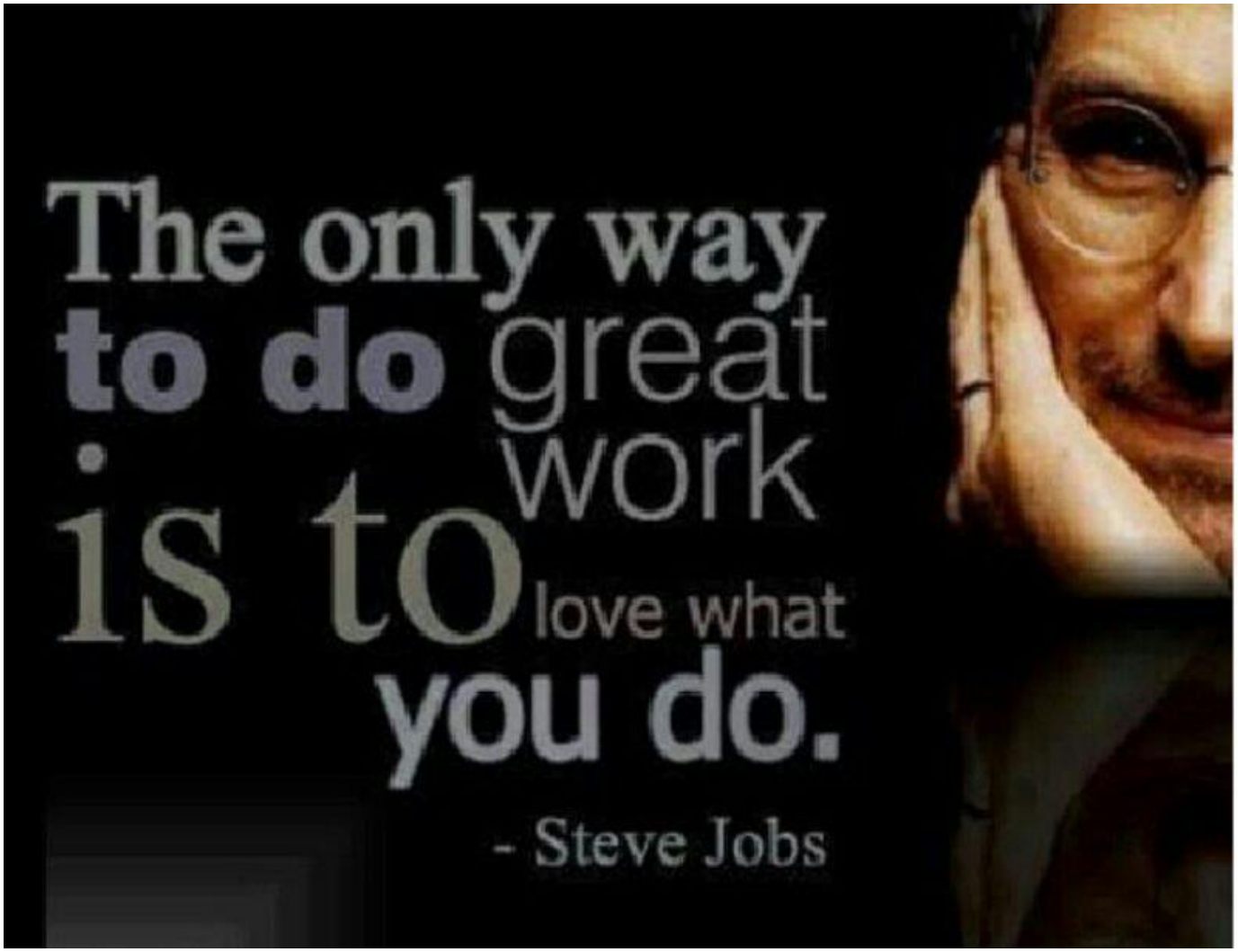
BIBLIOGRAFIA

Design Principles and Design Patterns
Robert C. Martin
http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf



Agile Principles, Patterns, and Practices in C#
Martin, Micah
2006 – Prentice Hall

PARA PENSAR



Elsa Estevez
ece@cs.uns.edu.ar