Principios del Diseño Básicos Orientado a Objetos

"Así como conocer las reglas del ajedrez no te hace un buen jugador, lo mismo pasa con la programación orientada a objetos: se puede conocer los conceptos pero no saber utilizarlos."
(Adrian Paredes)

¿Te consideras hijo del Drag and Drop? (Construir productos para entregar al cliente, de la forma más rápida posible). Que ha pasado con la Ingeniería de Software?

Inicial	Significado	Concepto
S	SRP	Principio de Única Responsabilidad (Single responsibility principle): *A class should have one, and only one, reason to change*.
O	OCP	Principio Abierto/Cerrado: *You should be able to extend a classes behavior, without modifying it*.
L	LSP	Principio de sustitución de Liskov: *Derived classes must be substitutable for their base classes.* Ver también diseño por contrato.
I	ISP	Principio de Segregación de la Interface (Interface segregation principle): *Make fine grained interfaces that are client specific.*
D	DIP	Principio de Inversión de Dependencia (Dependency inversion principle): Depend on abstractions, not on concretions. La Inyección de Dependencias es una de los métodos que siguen este principio.

SRP: THE SINGLE RESPONSIBILITY PRINCIPLE

(Fuente: https://docs.google.com/file/d/0ByOwmqah_nuGNHEtcU5OekdDMkk/edit?pli=1)

This principle was described in the work of Tom DeMarco and Meilir Page-Jones. They called it cohesion. They defined cohesion as the functional relatedness of the elements of a module. In this chapter we'll shift that meaning a bit, and relate cohesion to the forces that cause a module, or a class, to change.

"A CLASS SHOULD HAVE ONLY ONE REASON TO CHANGE."

Consider the bowling game. For most of its development the Game class was handling two separate responsibilities. It was keeping track of the current frame, and it was calculating the score. In the end, RCM and RSK separated these two responsibilities into two classes. The Game kept the responsibility to keep track of frames, and the Scorer got the responsibility to calculate the score.

Why was it important to separate these two responsibilities into separate classes? Because each responsibility is an axis of change. When the requirements change, that change will be manifest through a change in responsibility amongst the classes. If a class assumes more than one responsibility, then there will be more than one reason for it to change.

If a class has more than one responsibility, then the responsibilities become coupled. Changes to one responsibility may impair or inhibit the class' ability to meet the others. This kind of coupling leads to fragile designs that break in unexpected ways when changed.

For example, consider the design in Figure 1. The Rectangle class has two methods shown. One draws the rectangle on the screen, the other computes the area of the rectangle.

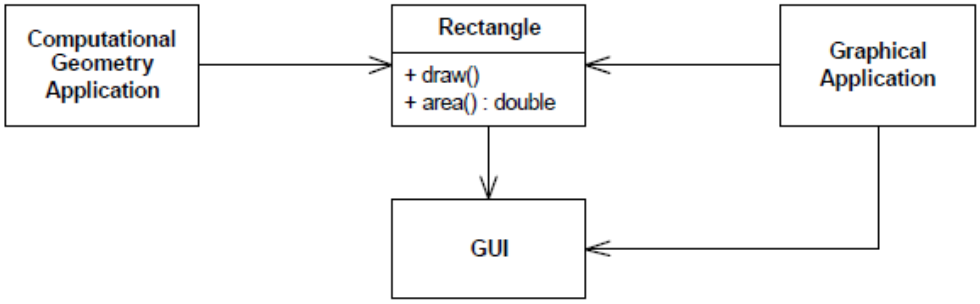

Figure 1

Two different applications use the Rectangle class. One application does computational geometry. It uses Rectangle to help it with the mathematics of geometric shapes. It never draws the rectangle on the screen. The other application is graphical in nature. It may also do some computational geometry, but it definitely draws the rectangle on the screen.

This design violates the SRP. The Rectangle class has two responsibilities. The first responsibility is to provide a mathematical model of the geometry of a rectangle. The second responsibility is to render the rectangle on a graphical user interface. The violation of SRP causes several nasty problems. Firstly, we must include the GUI in the computational geometry application. In .NET the GUI assembly would have to be built and deployed with the computational geometry application. Secondly, if a change to the GraphicalApplication causes the Rectangle to change for some reason, that change may force us to rebuild, retest, and redeploy the ComputationalGeometryApplication. If we forget to do this, that application may break in unpredictable ways.

A better design is to separate the two responsibilities into two completely different classes as shown in Figure 2. This design moves the computational portions of Rectangle into the GeometricRectangle class. Now changes made to the way rectangles are rendered cannot affect the ComputationalGeometryApplication.

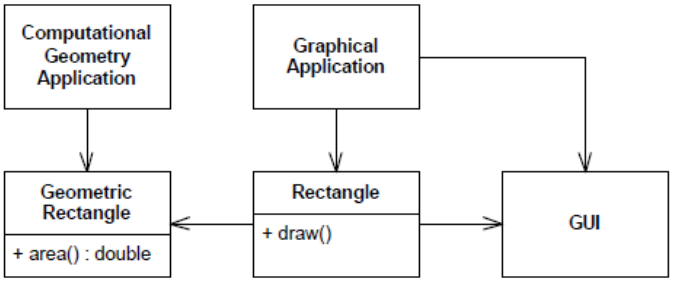

Figure 2

What is a Responsibility?

In the context of the Single Responsibility Principle (SRP) we define a responsibility to be "a reason for change." If you can think of more than one motive for changing a class, then that class has more than one responsibility. This is sometimes hard to see. We are accustomed to thinking of responsibility in groups. For example, consider the Modem interface in Listing 1. Most of us will agree that this interface looks perfectly reasonable. The four functions it declares are certainly functions belonging to a modem.

```
public interface Modem{
public void Dial(string pno);
public void Hangup();
public void Send(char c);
public char Recv();
}
```

Listing 1

However, there are two responsibilities being shown here. The first responsibility is connection management. The second is data communication. The dial and hangup functions manage the connection of the modem, while the send and recv functions communicate data. Should these two responsibilities be separated? That depends upon how the application is changing. If the application changes in ways that affect the signature of the connection functions, then the design will smell of Rigidity because the classes that call send and read will have to be recompiled and redeployed more often than we like. In that case the two responsibilities should be separated as shown in Figure 3. This keeps the client applications from coupling the two responsibilities.

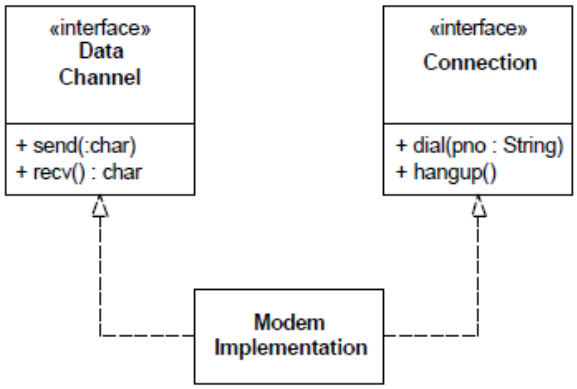

Figure 3

If, on the other hand, the application is not changing in ways that cause the the two responsibilities to change at differen times, then there is no need to separate them. Indeed, separating them would smell of Needless Complexity. There is a corrolary here. *An axis of change is only an axis of change if the changes actually occurr*. It is not wise to apply the SRP, or any other principle for that matter, if there is no symptom.

Separating coupled responsibilities.

Notice that in Figure 3 I kept both responsibilities coulped in the ModemImplementation class. This is not desirable, but it may be necessary. There are often reasons, having to do with the details of the hardware or OS, that force us to couple things that we'd rather not couple.

However, by separating their interfaces we have decoupled the concepts as far as the rest of the application is concerned.

We may view the ModemImplementation class as a kludge, or a wart; however, notice that all dependencies flow *away* from it. Nobody need depend upon this class.

Nobody except main needs to know that it exists. Thus, we've put the ugly bit behind a fence. It's ugliness need not leak out and pollute the rest of the application.

Persistence.
Figure 4 shows a common violation of the SRP. The Employee class contains business rules and persistence control. These two responsibilities should almost never be mixed. Business rules tend to change frequently, and though persistence may not change as frequently, it changes for completely different reasons. Binding business rules to the persistence subsystem is asking for trouble.

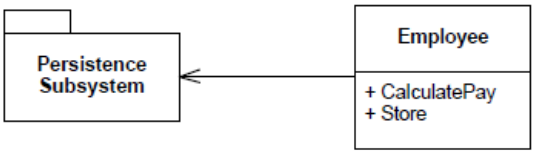

Figure 4

Fortunately, the practice of test driven development will usually force these two responsibilities to be separated long before the design begins to smell. However, in cases where the tests did not force the separation, and the smells of Rigidity and Fragility become strong, the design should be refactored using the FACADE, DAO, or PROXY patterns to separate the two responsibilities.

Ejemplo No 1:

a. Se tiene una clase Pedido con dos métodos: cargar, que traería de la base de datos la información del pedido y de los productos que lo componen, y calcularCosto, que sumaría el precio de los productos, aplicando cualquier descuento pertinente.

Esto podría parecer un buen diseño, porque, al fin y acabo, ambos métodos se encargan de cosas relacionadas con los pedidos, pero la realidad es que la clase se encarga de tareas relacionadas con los pedidos y de tareas relacionadas con la base de datos