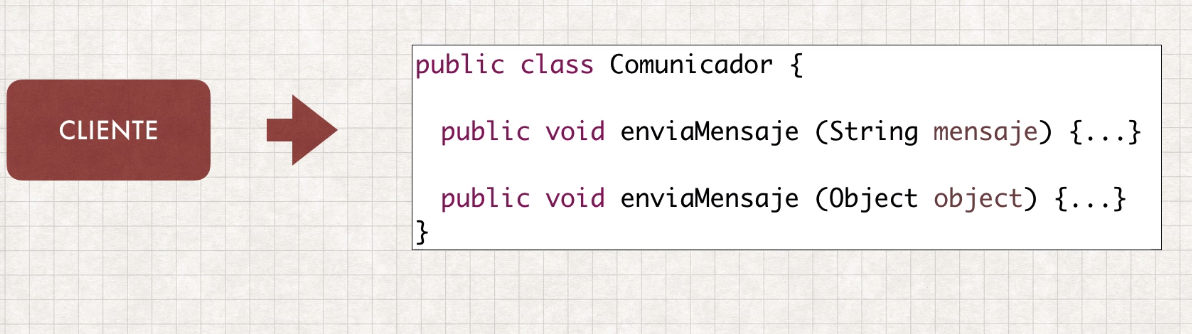
**1.4. Interface Segregation Principle:**

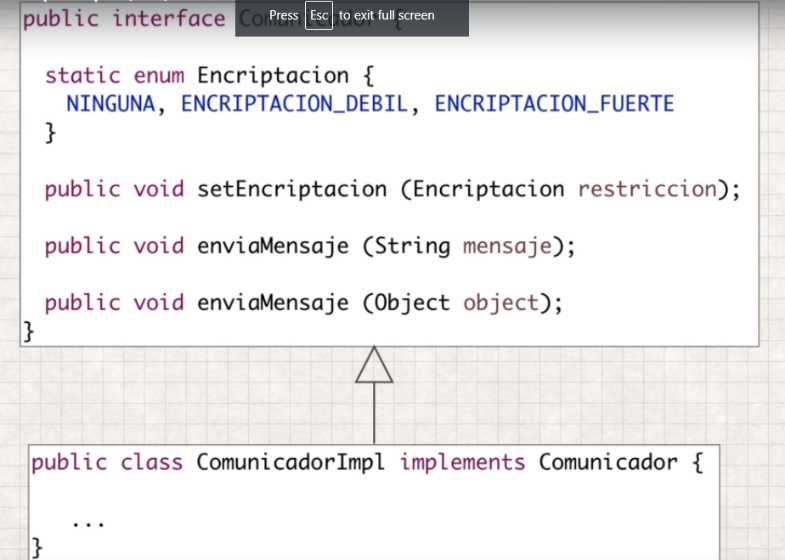
The clients should not be forced to depend on methods they do not use.

**- Example:**



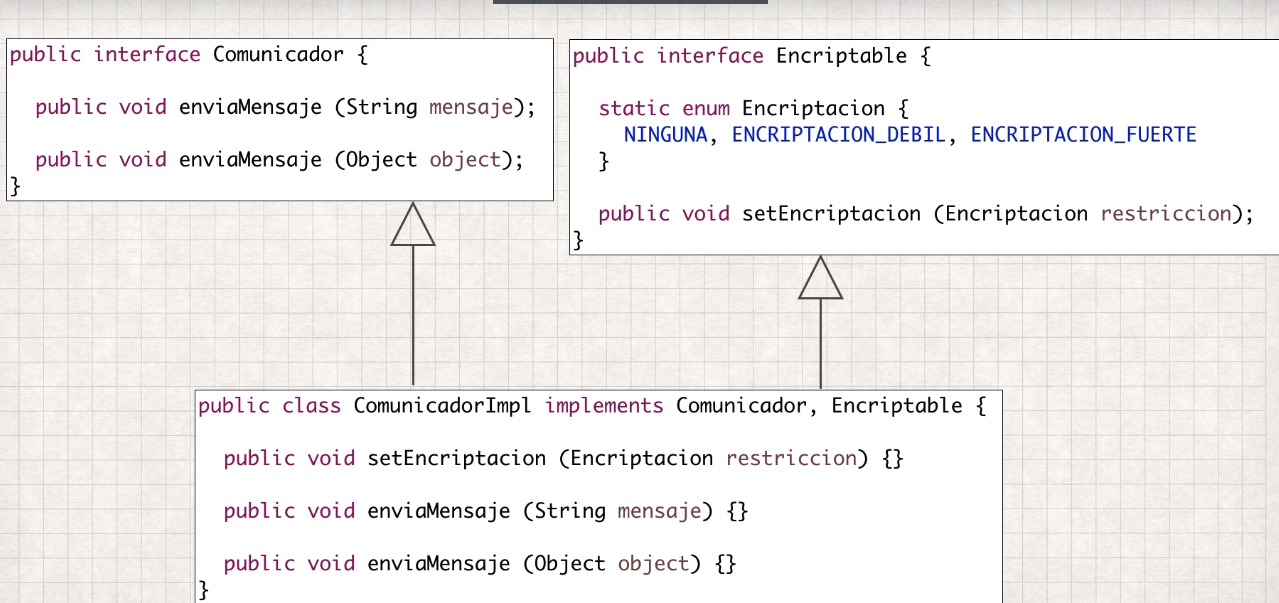
We have a system to send messages that is being used by several clients. Our Communicator class has 2 methods, sendMessage(String message) and sendMessage(Object message).

One day, 1 client request us to add a new feature to set the message encrypt, so, we proceed to implement it in the next way:



**- Problem:** the problem of the current implementation is that not all the clients requires the setEncrypt() method, so if the “Communicator”class is implementing the Comunicador Interface, then all the Communicators will have the setEncrypt() method, also when they do not needed.

**- Solution:**



If we split the functionality creating two different interfaces, then every communicator will be able to implement the required interfaces having the methods that everyone needs. This solution also follow the Single Responsibility Principle.

**Segregación de interfaces:**

El principio de segregación de interfaces viene a decir que **ninguna clase debería depender de métodos que no usa**. Por tanto, cuando creemos interfaces que definan comportamientos, es importante estar seguros de que todas las clases que implementen esas interfaces vayan a necesitar y ser capaces de agregar comportamientos a todos los métodos. **En caso contrario, es mejor tener varias interfaces más pequeñas**.

**Las interfaces nos ayudan a desacoplar módulos entre sí**. Esto es así porque si tenemos una interfaz que explica el comportamiento que el módulo espera para comunicarse con otros módulos, nosotros siempre podremos crear una clase que lo implemente de modo que cumpla las condiciones.

El módulo que describe la interfaz no tiene que saber nada sobre nuestro código y, sin embargo, nosotros podemos trabajar con él sin problemas.

### El problema

La problemática surge cuando esas interfaces intentan definir más cosas de las debidas, lo que se denominan fat interfaces.

Probablemente ocurrirá que **las clases hijas acabarán por no usar muchos de esos métodos**, y habrá que darles una implementación.

Muy habitual es lanzar una excepción, o simplemente no hacer nada.

Pero, al igual que vimos en algún ejemplo en el principio de sustitución de Liskov, **esto es peligroso**. Si lanzamos una excepción, es más que probable que el módulo que define esa interfaz use el método en algún momento, y esto hará fallar nuestro programa.

El resto de implementaciones «por defecto» que podamos dar, pueden generar efectos secundarios que no esperemos, y a los que sólo podemos responder conociendo el código fuente del módulo en cuestión, cosa que no nos interesa.

### ¿Cómo detectar que estamos violando el Principio de segregación de interfaces?

Como comentaba en los párrafos anteriores, si **al implementar una interfaz ves que uno o varios de los métodos no tienen sentido y te hace falta dejarlos vacíos o lanzar excepciones**, es muy probable que estés violando este principio.

Si la interfaz forma parte de tu código, divídela en varias interfaces que definan comportamientos más específicos.

Recuerda que no pasa nada porque una clase ahora necesite implementar varias interfaces. El punto importante es que use todos los métodos definidos por esas interfaces.

**Ejemplo:**

Imagina que tienes una tienda de CDs de música, y que tienes modelados tus productos de esta manera:

**interface** Product {

**val** name: **String**

**val** stock: **Int**

**val** numberOfDisks: **Int**

**val** releaseDate: **Int**

}

**class** CD : Product {

...

}

El producto tiene una serie de propiedades que nuestra clase CD sobrescribirá de algún modo.

Pero ahora has decidido ampliar mercado, y empezar a vender DVDs también.

El problema es que para los DVDs necesitas almacenar también la clasificación por edades, porque tienes que asegurarte de que no vendes películas no adecuadas según la edad del cliente.

Lo más directo sería simplemente añadir la nueva propiedad a la interfaz:

**interface** Product {

...

**val** recommendedAge: **Int**

}

¿Qué ocurre ahora con los CDs? Que se ven obligados a implementar recommendedAge, pero no van a saber qué hacer con ello, así que lanzarán una excepción:

**class** CD : Product {

...

override **val** recommendedAge: **Int**

**get**() = **throw** UnsupportedOperationException()

}

Con todos los problemas asociados que hemos visto antes.

Además, se forma una dependencia muy fea, en la que cada vez que añadimos algo a Product, nos vemos obligados a modificar CD con cosas que no necesita.

**Podríamos hacer algo tal que así:**

**interface** DVD : Product {

**val** recommendedAge: **Int**

}

Y hacer que nuestras clases extiendan de aquí.

Esto solucionaría el problema a corto plazo, pero hay algunas cosas que pueden seguir sin funcionar demasiado bien.

Por ejemplo, si hay otro producto que necesite categorización por edades, necesitaremos repetir parte de esta interfaz.

Además, esto no nos permitiría realizar operaciones comunes a productos que tengan esta característica.

La alternativa es segregar las interfaces, y que cada clase utilice las que necesite. Tendríamos por tanto una interfaz nueva:

**interface** AgeAware {

**val** recommendedAge: **Int**

}

Y ahora nuestra clase DVD implementará las dos interfaces:

**class** CD : Product {

. ..

}

**class** DVD : Product, AgeAware {

...

}

La ventaja de esta solución es que ahora podemos tener código AgeAware, y todas las clases que implementen esta interfaz podrían participar en código común.

Imagina que no vendes sólo productos, sino también actividades, que necesitarían una interfaz diferente.

Estas actividades también podrían implementar la interfaz AgeAware, y podríamos tener código como el siguiente, independientemente del tipo de producto o servicio que vendamos:

**fun** checkUserCanBuy(user: User, ageAware: AgeAware)

= user.age >= ageAware.recommendedAge

**Que hacer con el código antiguo?**

Si ya tienes código que utiliza fat interfaces, la solución puede ser **utilizar el patrón de diseño «Adapter»**. El [patrón Adapter](https://es.wikipedia.org/wiki/Adapter_(patr%C3%B3n_de_dise%C3%B1o)) nos permite convertir unas interfaces en otras, por lo que puedes usar adaptadores que conviertan la interfaz antigua en las nuevas.

**Conclusión:**

El principio de segregación de interfaces nos ayuda a **no obligar a ninguna clase a implementar métodos que no utiliza**. Esto nos evitará problemas que nos pueden llevar a errores inesperados y a dependencias no deseadas. Además, nos ayuda a reutilizar código de forma más inteligente.