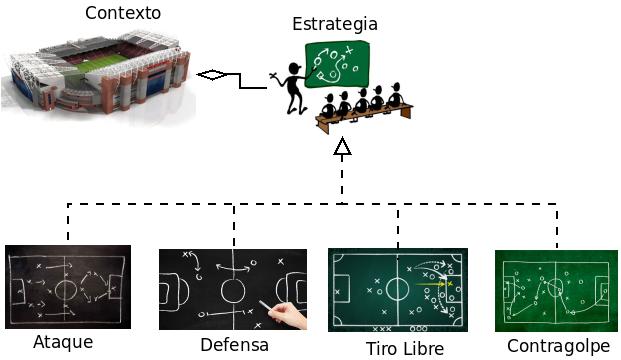
Imagina que eres el director técnico de un equipo de fútbol, y tienes un juego muy importante el próximo domingo, ¿cómo plantearías el juego?.

Como eres un buen director técnico, tienes que prever los posibles escenarios y tener una ***estrategia*** preparada para cada ***contexto***.

De esa manera la formación y/o ***comportamiento*** de tus jugadores deberían cambiar dependiendo de los diversos factores que se presenten en el desarrollo juego, tales como: cuando atacas, cuando defiendes, expulsan a un jugador, vas ganando, vas perdiendo, juegan de visita, juegan de local, el rival es débil, el rival es fuerte, etc.



interface *IAve* {

void volar();

void comer();

}

class Loro implements *IAve*{

@Override

public void volar() {

//...

}

@Override

public void comer() {

//..

}

}

class Tucan implements *IAve*{

@Override

public void volar() {

//...

}

@Override

public void comer() {

//..

}

}

Hasta aquí todo bien. Pero ahora imaginemos que queremos añadir a los pingüinos. Estos son aves, pero además tienen la habilidad de nadar. Podríamos hacer esto:

interface *IAve* {

void volar();

void comer();

void nadar();

}

class Loro implements *IAve*{

@Override

public void volar() {

//...

}

@Override

public void comer() {

//...

}

@Override

public void nadar() {

//...

}

}

class Pinguino implements *IAve*{

@Override

public void volar() {

//...

}

@Override

public void comer() {

//...

}

@Override

public void nadar() {

//...

}

}

El problema es que el loro no nada, y el pingüino no vuela, por lo que tendríamos que añadir una excepción o aviso si se intenta llamar a estos métodos. Además, si quisiéramos añadir otro método a la interfaz IAve, tendríamos que recorrer cada una de las clases que la implementa e ir añadiendo la implementación de dicho método en todas ellas. Esto viola el principio de segregación de interfaz, ya que estas clases (los clientes) no tienen por qué depender de métodos que no usan.

Lo más correcto sería segregar más las interfaces, tanto como sea necesario. En este caso podríamos hacer lo siguiente:

interface *IAve* {

void comer();

}

interface *IAveVoladora* {

void volar();

}

interface *IAveNadadora* {

void nadar();

}

class Loro implements IAve, *IAveVoladora*{

@Override

public void volar() {

//...

}

@Override

public void comer() {

//...

}

}

class Pinguino implements IAve, *IAveNadadora*{

@Override

public void nadar() {

//...

}

@Override

public void comer() {

//...

}

}

Si dentro de tu clase haces uso intensivo de las condicionales if, else, swicth, case, eso quiere decir que tu clase tiene asignado muchos comportamientos y/o responsabilidades, lo cual suele ser un indicador de la necesidad aplicar el patrón Strategy (estrategia), para poder encapsular estos comportamientos y delegarlos a otra clase u objeto.