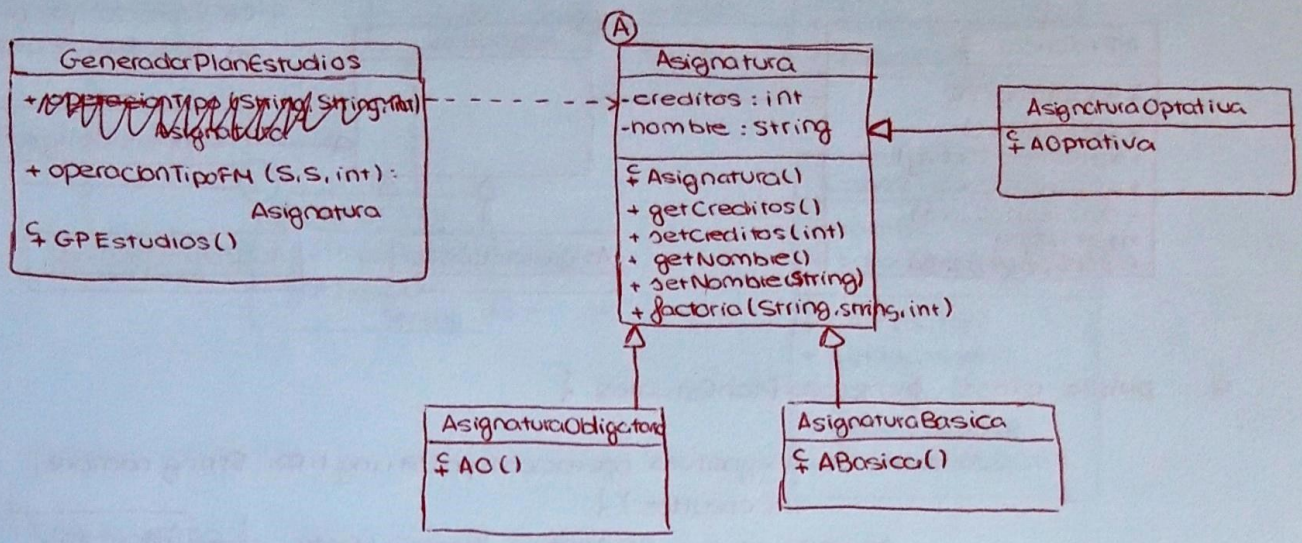
Patrones - Asignaturas

1 En este caso necesitamos un patroñ de tipo creacional, y en concreto un Factory Method, que nos permita desacoplar la creación del objeto de como se va a manejar. Usaremos la versión parametrizada, donde al método le pasaremos el tipo de asignatura. Un Abstract Factory no hace falta ya que solo tenemos un tipo de producto. La versión parametrizado es porque un alumno puede tener asignaturas de varios tipos.

2. Las clases siguen siendo las mismas, lo único que cambia es que en la clase Asignatura se añade el metodo factoria, que segun el tipo de asignatura, crea objetos de una subclase u otra.



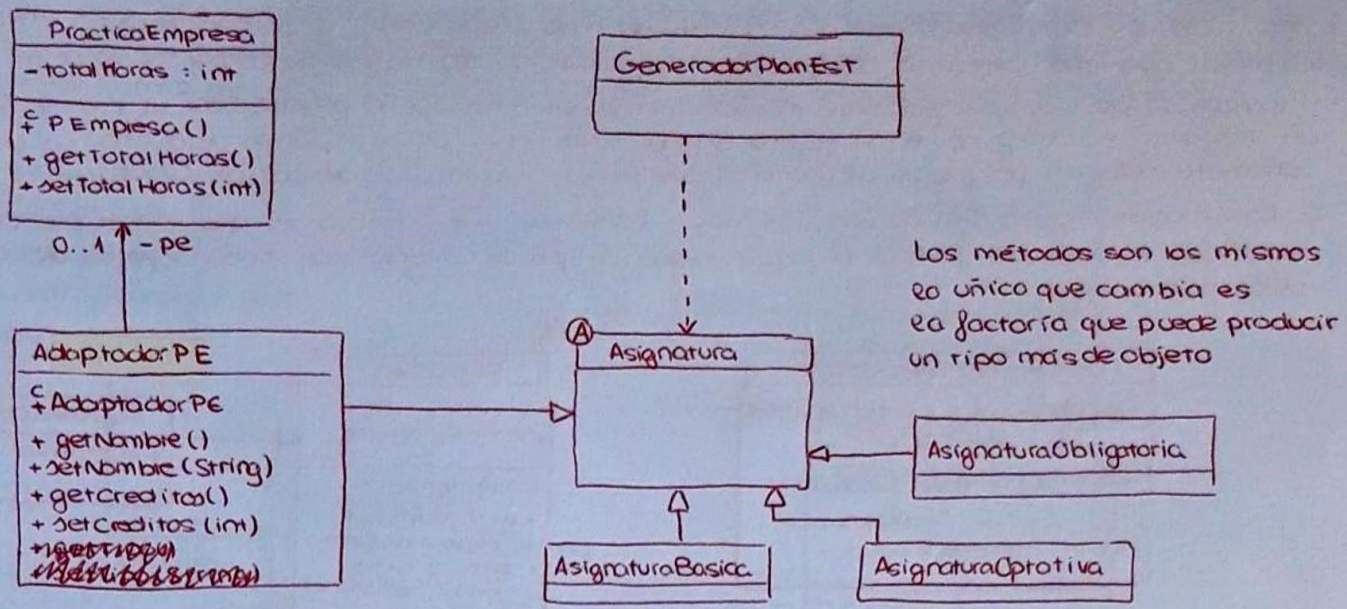
En la clase cliente se sustituye el método operacionTipo por la versión nueva que trabaja con la factoria.

3

```
public class GeneradorPlanEstudios {
    ...
    public Asignatura operacionTipo (String tipo , String nombre
                int créditos) {
        Asignatura a = Asignatura.factoria(tipo);
        a.SetNombre(nombre);
        a.setCreditos(creditos);
        //otras operaciones sobre a
        return a;
    }
//...
}
```

4. En este caso necesitamos un patroñ de tipo estructural, en concreto, un patroñ Adapter, que nos permita utilizar objetos de la clase PracticaEmpresa como si fueran del tipo Asignatura. En este caso se podria usar tanto el adaptador de clases como el de objetos (usaremos este) respetando que la clase PracticaEmpresa no puede ser modificada.

5. Habria que crear la clase Adaptador, AdaptadorPracticaEmpresa, que hereda de Asignatura y utiliza por composición PracticaEmpresa en un atributo. Se sobreescriben los métodos de Asignatura en AdaptadorPracticaEmpresa usando los métodos del atributo. La Aplicación cliente manipula indistintamente los dos tipos de objetos ya que por herencia AdaptadorPracticaEmpresa es un subtipo de Asignatura.



6.
```
public class GeneradorPlanEstudios {
    ...
    public static Asignatura operacionTipo(String tipo, String nombre,
                                           int creditos) {
        Asignatura a = Asignatura.factoria(tipo);
        a.setNombre(nombre);
        a.setCreditos(creditos);
        // resto de operaciones
        return a;
    }
}
```

```
Factoria
switch(tipo){
...
case "empresa":
return new
AdaptadorPracticaE();
...
```

new AdaptadorPE());

7. En este caso necesitamos un patrón de tipo creacional, y en concreto el patrón singleton, Nos ofrece una solución a cada problema planteado:

- Por medio de su instancia privada, Singleton permite utilizar siempre el mismo objeto de la clase PracticaEmpresa. Además, su inicialización perezosa permite crear el objeto solo cuando se necesita.
- Al ofrecer acceso mediante métodos estáticos, estarán disponibles desde cualquier punto del código sin tener que recibir como argumento el objeto en cuestión en cada método que se necesite.
- Al acceder siempre a la misma instancia, sus atributos se conservan entre usos y no perdemos el total de horas. Además el bloquear el new con constructores privados evita la creación de instancias por error.